SYBASE iAnywhere

SQL Anywhere® Server
SQL Reference

Published: March 2007

Copyright and trademarks

Contents

About This Manual xi

SQL Anywhere documentation xii
Documentation conventions xv
Finding out more and providing feedback xix

I. Using SQL 1

SQL Language Elements 3
Keywords 4
Identifiers 7
Strings 8
Constants 9
Operators 11
Expressions 15
Search conditions 20
Special values 30
Variables 36
Comments 42
NULL value 43
SQL Data Types 47
Character data types 48
Numeric data types 56
Money data types 64
Bit array data types 65
Date and time data types 67
Binary data types 74
Domains 78
Data type conversions 80
Java and SQL data type conversion 88
SQL Functions 91
Introduction to SQL functions 92
Function types 93
Alphabetical list of functions 103

SQL Statements 289
Using the SQL statement reference 295
ALLOCATE DESCRIPTOR statement [ESQL] 299
ALTER DATABASE statement 301
ALTER DBSPACE statement 305
ALTER DOMAIN statement 307
ALTER EVENT statement 308
ALTER FUNCTION statement 310
ALTER INDEX statement 311
ALTER MATERIALIZED VIEW statement 313
ALTER PROCEDURE statement 315
ALTER PUBLICATION statement [MobiLink] [SQL Remote] 317
ALTER REMOTE MESSAGE TYPE statement [SQL Remote] 319
ALTER SERVER statement 321
ALTER SERVICE statement 323
ALTER STATISTICS statement 327
ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 328
ALTER SYNCHRONIZATION USER statement [MobiLink] 330
ALTER TABLE statement 332
ALTER TRIGGER statement 341
ALTER VIEW statement 342
ATTACH TRACING statement 344
BACKUP statement 346
BEGIN statement 351
BEGIN TRANSACTION statement [T-SQL] 354
BREAK statement [T-SQL] 356
CALL statement 357
CASE statement 359
CHECKPOINT statement 361
CLEAR statement [Interactive SQL] 362
CLOSE statement [ESQL] [SP] 363
COMMENT statement 365
COMMIT statement 367
CONFIGURE statement [Interactive SQL] 369
CONNECT statement [ESQL] [Interactive SQL] 370

CONTINUE statement [T-SQL] 373
CREATE DATABASE statement 374
CREATE DBSPACE statement 382
CREATE DECRYPTED FILE statement 384
CREATE DOMAIN statement 386
CREATE ENCRYPTED FILE statement 388
CREATE EVENT statement 390
CREATE EXISTING TABLE statement 395
CREATE EXTERNLOGIN statement 397
CREATE FUNCTION statement 399
CREATE INDEX statement 405
CREATE LOCAL TEMPORARY TABLE statement 409
CREATE MATERIALIZED VIEW statement 411
CREATE MESSAGE statement [T-SQL] 413
CREATE PROCEDURE statement 414
CREATE PROCEDURE statement [T-SQL] 425
CREATE PUBLICATION statement [MobiLink] [SQL Remote] 427
CREATE REMOTE MESSAGE TYPE statement [SQL Remote] 431
CREATE SCHEMA statement 433
CREATE SERVER statement 435
CREATE SERVICE statement 438
CREATE STATISTICS statement 442
CREATE SUBSCRIPTION statement [SQL Remote] 443
CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 445
CREATE SYNCHRONIZATION USER statement [MobiLink] 448
CREATE TABLE statement 450
CREATE TRIGGER statement 462
CREATE TRIGGER statement [T-SQL] 468
CREATE VARIABLE statement 469
CREATE VIEW statement 471
DEALLOCATE statement 474
DEALLOCATE DESCRIPTOR statement [ESQL] 475
Declaration section [ESQL] 476
DECLARE statement 477
DECLARE CURSOR statement [ESQL] [SP] 478

DECLARE CURSOR statement [T-SQL] 482
DECLARE LOCAL TEMPORARY TABLE statement 483
DELETE statement 485
DELETE (positioned) statement [ESQL] [SP] 488
DESCRIBE statement [ESQL] 490
DESCRIBE statement [Interactive SQL] 494
DETACH TRACING statement 496
DISCONNECT statement [ESQL] [Interactive SQL] 497
DROP statement 498
DROP CONNECTION statement 500
DROP DATABASE statement 501
DROP EXTERNLOGIN statement 502
DROP PUBLICATION statement [MobiLink] [SQL Remote] 503
DROP REMOTE MESSAGE TYPE statement [SQL Remote] 504
DROP SERVER statement 505
DROP SERVICE statement 506
DROP STATEMENT statement [ESQL] 507
DROP STATISTICS statement 508
DROP SUBSCRIPTION statement [SQL Remote] 509
DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 510
DROP SYNCHRONIZATION USER statement [MobiLink] 511
DROP VARIABLE statement 512
EXCEPT statement 513
EXECUTE statement [ESQL] 515
EXECUTE statement [T-SQL] 517
EXECUTE IMMEDIATE statement [SP] 519
EXIT statement [Interactive SQL] 522
EXPLAIN statement [ESQL] 524
FETCH statement [ESQL] [SP] 526
FOR statement 530
FORWARD TO statement 533
FROM clause 535
GET DATA statement [ESQL] 542
GET DESCRIPTOR statement [ESQL] 544
GET OPTION statement [ESQL] 546

GOTO statement [T-SQL] 547
GRANT statement 548
GRANT CONSOLIDATE statement [SQL Remote] 553
GRANT PUBLISH statement [SQL Remote] 555
GRANT REMOTE statement [SQL Remote] 556
GRANT REMOTE DBA statement [MobiLink] [SQL Remote] 558
GROUP BY clause 559
HELP statement [Interactive SQL] 562
IF statement 563
IF statement [T-SQL] 565
INCLUDE statement [ESQL] 567
INPUT statement [Interactive SQL] 568
INSERT statement 573
INSTALL JAVA statement 578
INTERSECT statement 580
LEAVE statement 582
LOAD STATISTICS statement 584
LOAD TABLE statement 585
LOCK TABLE statement 593
LOOP statement 595
MESSAGE statement 597
OPEN statement [ESQL] [SP] 601
OUTPUT statement [Interactive SQL] 604
PARAMETERS statement [Interactive SQL] 608
PASSTHROUGH statement [SQL Remote] 609
PREPARE statement [ESQL] 610
PREPARE TO COMMIT statement 612
PRINT statement [T-SQL] 613
PUT statement [ESQL] 614
RAISERROR statement [T-SQL] 616
READ statement [Interactive SQL] 618
READTEXT statement [T-SQL] 620
REFRESH MATERIALIZED VIEW statement 621
REFRESH TRACING LEVEL statement 623
RELEASE SAVEPOINT statement 625

REMOTE RESET statement [SQL Remote] 626
REMOVE JAVA statement 627
REORGANIZE TABLE statement 628
RESIGNAL statement 630
RESTORE DATABASE statement 631
RESUME statement 633
RETURN statement 634
REVOKE statement 636
REVOKE CONSOLIDATE statement [SQL Remote] 638
REVOKE PUBLISH statement [SQL Remote] 639
REVOKE REMOTE statement [SQL Remote] 640
REVOKE REMOTE DBA statement [SQL Remote] 641
ROLLBACK statement 642
ROLLBACK TO SAVEPOINT statement 643
ROLLBACK TRANSACTION statement [T-SQL] 644
ROLLBACK TRIGGER statement 645
SAVE TRANSACTION statement [T-SQL] 646
SAVEPOINT statement 647
SELECT statement 648
SET statement 656
SET statement [T-SQL] 658
SET CONNECTION statement [Interactive SQL] [ESQL] 661
SET DESCRIPTOR statement [ESQL] 662
SET OPTION statement 664
SET OPTION statement [Interactive SQL] 667
SET REMOTE OPTION statement [SQL Remote] 668
SET SQLCA statement [ESQL] 670
SETUSER statement 671
SIGNAL statement 673
START DATABASE statement 674
START ENGINE statement [Interactive SQL] 676
START JAVA statement 677
START LOGGING statement [Interactive SQL] 678
START SUBSCRIPTION statement [SQL Remote] 679
START SYNCHRONIZATION DELETE statement [MobiLink] 681

STOP DATABASE statement 683
STOP ENGINE statement 684
STOP JAVA statement 685
STOP LOGGING statement [Interactive SQL] 686
STOP SUBSCRIPTION statement [SQL Remote] 687
STOP SYNCHRONIZATION DELETE statement [MobiLink] 688
SYNCHRONIZE SUBSCRIPTION statement [SQL Remote] 689
SYSTEM statement [Interactive SQL] 691
TRIGGER EVENT statement 692
TRUNCATE TABLE statement 693
UNION statement 695
UNLOAD statement 698
UNLOAD TABLE statement 700
UPDATE statement 703
UPDATE (positioned) statement [ESQL] [SP] 708
UPDATE statement [SQL Remote] 710
VALIDATE statement 713
WAITFOR statement 715
WHENEVER statement [ESQL] 717
WHILE statement [T-SQL] 718
WINDOW clause 719
WRITETEXT statement [T-SQL] 722

II. System Objects 723

Tables 725
System tables 726
Diagnostic tracing tables 735
Other tables 751
Views 753
System views in Sybase Central 754
Consolidated views 809
Compatibility views 824
System Procedures 833
Introduction to system procedures 834
System procedures 835

System extended procedures 951
Adaptive Server Enterprise system and catalog procedures 962
Index 965

About This Manual

Subject

This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.

While other manuals provide more motivation and context for how to carry out particular tasks, this manual is the place to look for complete listings of available SQL syntax and system objects.

Audience

This manual is for all users of SQL Anywhere. It includes material of particular interest to users of MobiLink, UltraLite and SQL Remote. It is to be used in conjunction with other manuals in the documentation set.

SQL Anywhere documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—a product that provides data management and data exchange technologies, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software, as well as upgrade instructions.
- **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by these tools.
- **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink server-side utilities and functionality.
- **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- **MobiLink - Server-Initiated Synchronization** This manual describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.

- **QAnywhere** This manual describes QAnywhere, which is a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **SQL Anywhere 10 - Context-Sensitive Help** This manual contains the context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, MobiLink Model mode, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.
- **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ► Programs ► SQL Anywhere 10 ► Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on Unix operating systems, see the HTML documentation under your SQL Anywhere installation or on your installation CD.

- **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

On Windows, the PDF books are accessible from the online documentation via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are available on your installation CD.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, …]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

Operating system conventions

- **Windows** The Microsoft Windows family of operating systems for desktop and laptop computers. The Windows family includes Windows Vista and Windows XP.

- **Windows CE** Platforms built from the Microsoft Windows CE modular operating system, including the Windows Mobile and Windows Embedded CE platforms.

  Windows Mobile is built on Windows CE. It provides a Windows user interface and additional functionality, such as small versions of applications like Word and Excel. Windows Mobile is most commonly seen on mobile devices.

  Limitations or variations in SQL Anywhere are commonly based on the underlying operating system (Windows CE), and seldom on the particular variant used (Windows Mobile).

- **Unix** Unless specified, Unix refers to both Linux and Unix platforms.

File name conventions

The documentation generally adopts Windows conventions when describing operating system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

  ```
  MobiLink\redirector
  ```

  On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

  ```
  MobiLink/redirector
  ```

  If SQL Anywhere is used in a multi-platform environment you must be aware of path name differences between platforms.

- **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

  For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

- **install-dir** The installation process allows you to choose where to install SQL Anywhere, and the documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY10 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). SQLANYSH10 specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

  For more information on the default location of *install-dir*, by operating system, see "SQLANY10 environment variable" [*SQL Anywhere Server - Database Administration*].

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

After installation is complete, the environment variable SQLANYSAMP10 specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ► SQL Anywhere 10 ► Sample Applications and Projects opens a Windows Explorer window in this directory.

For more information on the default location of *samples-dir*, by operating system, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$envvar* or *${envvar}*.

  Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

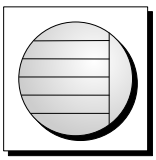
Graphic icons

The following icons are used in this documentation.

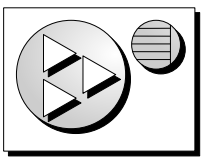
- A client application.

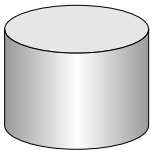
- A database server, such as SQL Anywhere.

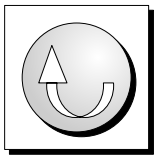
- An UltraLite application.

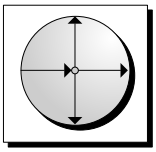
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.



- A Sybase Replication Server



- A programming interface.

Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at http://www.ianywhere.com/developer/.

If you have questions or need help, you can post messages to the Sybase iAnywhere newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by entering **dbeng10 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

Newsgroup disclaimer
iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.
iAnywhere Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

Part I. Using SQL

This section describes the SQL Anywhere SQL language, including data types, functions, and statements.

CHAPTER 1

SQL Language Elements

Contents

Keywords 4
Identifiers 7
Strings 8
Constants 9
Operators 11
Expressions 15
Search conditions 20
Special values 30
Variables 36
Comments 42
NULL value 43

Keywords

Each SQL statement contains one or more keywords. SQL is case insensitive to keywords, but throughout these manuals, keywords are indicated in uppercase.

For example, in the following statement, SELECT and FROM are keywords:

```
SELECT *
FROM Employees;
```

The following statements are equivalent to the one above:

```
Select *
From Employees;
select * from Employees;
sELECT * FRoM Employees;
```

Some keywords cannot be used as identifiers without surrounding them in double quotes. These are called reserved words. Other keywords, such as DBA, do not require double quotes, and are not reserved words.

Reserved words

Some keywords in SQL are also **reserved words**. To use a reserved word in a SQL statement as an identifier, you must enclose it in double quotes. Many, but not all, of the keywords that appear in SQL statements are reserved words. For example, you must use the following syntax to retrieve the contents of a table named SELECT.

```
SELECT *
FROM "SELECT"
```

Because SQL is not case sensitive with respect to keywords, each of the following words may appear in uppercase, lowercase, or any combination of the two. All strings that differ only in capitalization from one of the following words are reserved words.

If you are using embedded SQL, you can use the sql_needs_quotes database library function to determine whether a string requires quotation marks. A string requires quotes if it is a reserved word or if it contains a character not ordinarily allowed in an identifier.

For more information, see “sql_needs_quotes function” [*SQL Anywhere Server - Programming*].

The reserved SQL keywords in SQL Anywhere are as follows:

add	**all**	**alter**	**and**
any	**as**	**asc**	**attach**
backup	**begin**	**between**	**bigint**
binary	**bit**	**bottom**	**break**
by	**call**	**capability**	**cascade**

case	**cast**	**char**	**char_convert**
character	**check**	**checkpoint**	**close**
comment	**commit**	**compressed**	**conflict**
connect	**constraint**	**contains**	**continue**
convert	**create**	**cross**	**cube**
current	**current_timestamp**	**current_user**	**cursor**
date	**dbspace**	**deallocate**	**dec**
decimal	**declare**	**default**	**delete**
deleting	**desc**	**detach**	**distinct**
do	**double**	**drop**	**dynamic**
else	**elseif**	**encrypted**	**end**
endif	**escape**	**except**	**exception**
exec	**execute**	**existing**	**exists**
externlogin	**fetch**	**first**	**float**
for	**force**	**foreign**	**forward**
from	**full**	**goto**	**grant**
group	**having**	**holdlock**	**identified**
if	**in**	**index**	**index_lparen**
inner	**inout**	**insensitive**	**insert**
inserting	**install**	**instead**	**int**
integer	**integrated**	**intersect**	**into**
iq	**is**	**isolation**	**join**
kerberos	**key**	**lateral**	**left**
like	**lock**	**login**	**long**
match	**membership**	**message**	**mode**
modify	**natural**	**nchar**	**new**
no	**noholdlock**	**not**	**notify**
null	**numeric**	**nvarchar**	**of**

off	**on**	**open**	**option**
options	**or**	**order**	**others**
out	**outer**	**over**	**passthrough**
precision	**prepare**	**primary**	**print**
privileges	**proc**	**procedure**	**publication**
raiserror	**readtext**	**real**	**reference**
references	**refresh**	**release**	**remote**
remove	**rename**	**reorganize**	**resource**
restore	**restrict**	**return**	**revoke**
right	**rollback**	**rollup**	**save**
savepoint	**scroll**	**select**	**sensitive**
session	**set**	**setuser**	**share**
smallint	**some**	**sqlcode**	**sqlstate**
start	**stop**	**subtrans**	**subtransaction**
synchronize	**syntax_error**	**table**	**temporary**
then	**time**	**timestamp**	**tinyint**
to	**top**	**tran**	**trigger**
truncate	**tsequal**	**unbounded**	**union**
unique	**uniqueidentifier**	**unknown**	**unsigned**
update	**updating**	**user**	**using**
validate	**values**	**varbinary**	**varbit**
varchar	**variable**	**varying**	**view**
wait	**waitfor**	**when**	**where**
while	**window**	**with**	**with_cube**
with_lparen	**with_rollup**	**within**	**work**
writetext	**xml**		

Identifiers

Identifiers are names of objects in the database, such as user IDs, tables, and columns.

Remarks

Identifiers have a maximum length of 128 bytes. They must be enclosed in double quotes or square brackets if any of the following conditions are true:

- The identifier contains spaces.
- The first character of the identifier is not an alphabetic character (as defined below).
- The identifier contains a reserved word.
- The identifier contains characters other than alphabetic characters and digits.

Alphabetic characters include the alphabet, as well as the underscore character (_), at sign (@), number sign (#), and dollar sign ($). The database collation sequence dictates which characters are considered alphabetic or digit characters.

The following characters are not permitted in identifiers:

- Double quotes
- Control characters (any character less than 0x20)
- Double backslashes

You can use a single backslash in an identifier only if it is used as an escape character.

If the quoted_identifier database option is set to Off, double quotes are used to delimit SQL strings and cannot be used for identifiers. However, you can always use square brackets to delimit identifiers, regardless of the setting of quoted_identifier. The default setting for the quoted_identifier option is to Off for Open Client and jConnect connections; otherwise the default is On.

See also

- For a complete list of the reserved words, see "Reserved words" on page 4.
- For information about the quoted_identifier option, see "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Examples

The following are all valid identifiers.

```
Surname
"Surname"
[Surname]
SomeBigName
"Client Number"
```


Strings

A string is a sequence of characters up to 2 GB in size. A string can occur in SQL:

- as a **string literal**. A string literal is a sequence of characters enclosed in single quotes (apostrophes). A string literal represents a particular, constant value, and it may contain escape sequences for special characters that cannot be easily typed as characters. See "Binary literals" on page 9.
- as the value of a column or variable with a CHAR or NCHAR data type.
- as the result of evaluating an expression.

The length of a string can be measured in two ways:

- **Byte length** The byte length is the number of bytes in the string.
- **Character length** The character length is the number of characters in the string, and is based on the character set being used.

For single-byte character sets, such as cp1252, the byte-length and character-length are the same. For multibyte character sets, a string's byte-length is greater than or equal to its character-length.

Constants

This section describes binary literals and string literals.

Binary literals

A binary literal is a sequence of hexadecimal characters consisting of digits 0-9 and uppercase and lowercase letters A-F. When you enter binary data as literals, you must precede the data by **0x** (a zero, followed by an x), and there should be an even number of digits to the right of this prefix. For example, the hexadecimal equivalent of 39 is 0027, and is expressed as 0x0027.

A binary literal is sometimes referred to as a binary constant. In SQL Anywhere, the preferred term is binary literal.

String literals

A string literal is a sequence of characters enclosed in single quotes. For example, `'Hello world'` is a string literal of type CHAR. Its byte length is 11, and its character length is also 11.

A string literal is sometimes referred to as a string constant, literal string, or just as a string. In SQL Anywhere, the preferred term is string literal.

You can specify an NCHAR string literal by prefixing the quoted value with N. For example, `N'Hello world'` is a string literal of type NCHAR. Its byte length is 11, and its character length is 11. The bytes within an NCHAR string literal are interpreted using the database's CHAR character set, and then converted to NCHAR. The syntax `N'`*string*`'` is a shortened form for `CAST( '`*string*`' AS NCHAR )`.

Escape sequences

Sometimes you need to put characters into string literals that cannot be typed or entered normally. Examples include control characters (such as a new line character), single quotes (which would otherwise mark the end of the string literal), and hexadecimal byte values. For this purpose, you use an escape sequence.

The following examples show how to use escape sequences in string literals.

- A single quote is used to mark the beginning and end of a string literal, so a single quote in a string must be escaped using an additional single quote, as follows:

  ```
  'John''s database'
  ```

- Hexadecimal escape sequences can be used for any character or binary value. A hexadecimal escape sequence is a backslash followed by an x followed by two hexadecimal digits. The hexadecimal value is interpreted as a character in the CHAR character set for both CHAR and NCHAR string literals. The following example, in code page 1252, represents the digits 1, 2, and 3, followed by the euro currency symbol.

  ```
  '123\x80'
  ```

- To represent a new line character, use a backslash followed by n (\n), as follows:


```
'First line:\nSecond line:'
```

- A backslash is used to mark the beginning of an escape sequence, so a backslash character in a string must be escaped using an additional backslash, as follows:

  ```
  'c:\\temp'
  ```

You can use the same characters and escape sequences with NCHAR string literals as with CHAR string literals.

If you need to use Unicode characters that cannot be typed directly into the string literal, use the UNISTR function. See "UNISTR function [String]" on page 272.

Operators

This section describes arithmetic, string, and bitwise operators. For information on comparison operators, see "Search conditions" on page 20.

The normal precedence of operations applies. Expressions in parentheses are evaluated first, then multiplication and division before addition and subtraction. String concatenation happens after addition and subtraction.

For more information, see "Operator precedence" on page 14.

Comparison operators

The syntax for comparison conditions is as follows:

expression compare expression

where *compare* is a comparison operator. The following comparison operators are available:

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to
<>	Not equal to
!>	Not greater than
!<	Not less than

Case sensitivity
All string comparisons are *case insensitive* unless the database was created as case sensitive.

Case sensitivity By default, SQL Anywhere databases are created as case insensitive. Comparisons are carried out with the same attention to case as the database they are operating on. You can control the case sensitivity of SQL Anywhere databases with the -c option when you create the database.

For more information about case sensitivity for string comparisons, see "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

Trailing blanks The behavior of SQL Anywhere when comparing strings is controlled by the -b option that is set when creating the database.

For more information about blank padding, see "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

Logical operators

Search conditions can be combined using AND, OR, and NOT.

Conditions are combined using AND as follows:

condition1 **AND** *condition2*

The combined condition is TRUE if both conditions are TRUE, FALSE if either condition is FALSE, and UNKNOWN otherwise.

Conditions are combined using OR as follows:

condition1 **OR** *condition2*

The combined condition is TRUE if either condition is TRUE, FALSE if both conditions are FALSE, and UNKNOWN otherwise.

The syntax for the NOT operator is as follows:

NOT *condition*

The NOT condition is TRUE if *condition* is FALSE, FALSE if *condition* is TRUE, and UNKNOWN if *condition* is UNKNOWN.

The IS operator provides a means to test a logical value. The syntax for the IS operator is as follows:

expression **IS** [**NOT**] *truth-value*

The condition is TRUE if the *expression* evaluates to the supplied *truth-value*, which must be one of TRUE, FALSE, UNKNOWN, or NULL. Otherwise, the value is FALSE.

For more information, see "Three-valued logic" on page 27.

Arithmetic operators

expression + expression Addition. If either expression is the NULL value, the result is NULL.

expression – expression Subtraction. If either expression is the NULL value, the result is NULL.

–expression Negation. If the expression is the NULL value, the result is NULL.

expression * expression Multiplication. If either expression is NULL, the result is NULL.

expression / expression Division. If either expression is NULL or if the second expression is 0, the result is NULL.

expression % expression Modulo finds the integer remainder after a division involving two whole numbers. For example, 21 % 11 = 10 because 21 divided by 11 equals 1 with a remainder of 10.

Standards and compatibility

- **Modulo** The % operator can be used in SQL Anywhere only if the percent_as_comment option is set to Off. The default value is On.

String operators

expression || expression String concatenation (two vertical bars). If either string is NULL, it is treated as the empty string for concatenation.

expression + expression Alternative string concatenation. When using the + concatenation operator, you must ensure the operands are explicitly set to character data types rather than relying on implicit data conversion.

For example, the following query returns the integer value **579:**

```
SELECT 123 + 456
```

whereas the following query returns the character string **123456:**

```
SELECT '123' + '456'
```

You can use the CAST or CONVERT function to explicitly convert data types.

Standards and compatibility

- **SQL/2003** The || operator is the SQL/2003 string concatenation operator.

Bitwise operators

The following operators can be used on integer data types and bit array data types in SQL Anywhere.

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
~	bitwise NOT

The bitwise operators &, | and ~ are not interchangeable with the logical operators AND, OR, and NOT.

Example

For example, the following statement selects rows in which the correct bits are set.

```
SELECT *
FROM tableA
WHERE (options & 0x0101) <> 0
```


Join operators

The SQL/2003 join syntax that uses a table expression in the FROM clause is supported. See "FROM clause" on page 535.

Support for Transact-SQL outer join operators *= and =* is deprecated. To use Transact SQL outer joins, the tsql_outer_joins database option must be set to On. See "tsql_outer_joins option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Operator precedence

The precedence of operators in expressions is as follows. The operators at the top of the list are evaluated before those at the bottom of the list.

1. unary operators (operators that require a single operand)
2. **&**, **|**, **^**, **~**
3. *******, **/**, **%**
4. **+**, **-**
5. **||**
6. **not**
7. **and**
8. **or**

When you use more than one operator in an expression, it is recommended that you make the order of operation explicit using parentheses.

Expressions

An expression is a statement that can be evaluated to return values.

Syntax

expression:
case-expression
| *constant*
| [*correlation-name*.]*column-name*
| **-** *expression*
| *expression operator expression*
| **(** *expression* **)**
| *function-name* **(** *expression*, … **)**
| *if-expression*
| *special value*
| **(** *subquery* **)**
| *variable-name*

Parameters

case-expression:
CASE *expression*
WHEN *expression*
THEN *expression*,…
[**ELSE** *expression*]
END

alternative form of case-expression:
CASE
WHEN *search-condition*
THEN *expression*, …
[**ELSE** *expression*]
END

constant:
integer | *number* | *string* | *host-variable*

special-value:
CURRENT { **DATE** | **TIME** | **TIMESTAMP** }
| **NULL**
| **SQLCODE**
| **SQLSTATE**
| **USER**

if-expression:
IF *condition*
THEN *expression*
[**ELSE** *expression*]
ENDIF

operator:
{ **+** | **-** | ***** | **/** | **||** | **%** }

Remarks

Expressions are used in many different places.

Expressions are formed from several different kinds of elements. These are discussed in the sections on functions and variables. See "SQL Functions" on page 91, and "Variables" on page 36.

You must be connected to the database in order evaluate expressions.

Side effects

None.

See also

- "Constants in expressions" on page 16
- "Special values" on page 30
- "Column names in expressions" on page 16
- "SQL Functions" on page 91
- "Subqueries in expressions" on page 16
- "Search conditions" on page 20
- "SQL Data Types" on page 47
- "Variables" on page 36
- "CASE expressions" on page 17

Standards and compatibility

- For other differences, see the separate descriptions of each class of expression, in the following sections.

Constants in expressions

Constants are numbers or string literals. String constants are enclosed in apostrophes ('single quotes'). An apostrophe is represented inside a string by two apostrophes in a row.

Column names in expressions

A column name is an identifier preceded by an optional correlation name. (A correlation name is usually a table name. For more information on correlation names, see "FROM clause" on page 535.) If a column name has characters other than letters, digits and underscore, it must be surrounded by quotation marks (""). For example, the following are valid column names:

```
Employees.Name
address
"date hired"
"salary"."date paid"
```

For more information on identifiers, see "Identifiers" on page 7.

Subqueries in expressions

A subquery is a SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

The SELECT statement must be enclosed in parentheses, and must contain one and only one select list item. When used as an expression, a subquery is generally allowed to return only one value.

A subquery can be used anywhere that a column name can be used. For example, a subquery can be used in the select list of another SELECT statement.

For other uses of subqueries, see "Subqueries in search conditions" on page 21.

IF expressions

The syntax of the IF expression is as follows:

IF *condition*
THEN *expression1*
[**ELSE** *expression2*]
ENDIF

This expression returns the following:

- If *condition* is TRUE, the IF expression returns *expression1*.
- If *condition* is FALSE, the IF expression returns *expression2*.
- If *condition* is FALSE, and there is no *expression2*, the IF expression returns NULL.
- If condition is UNKNOWN, the IF expression returns NULL.

For more information about TRUE, FALSE and UNKNOWN conditions, see "NULL value" on page 43, and "Search conditions" on page 20.

IF statement is different from IF expression
Do not confuse the syntax of the IF expression with that of the IF statement. For information on the IF statement, see "IF statement" on page 563.

CASE expressions

The CASE expression provides conditional SQL expressions. Case expressions can be used anywhere an expression can be used.

The syntax of the CASE expression is as follows:

CASE *expression*
WHEN *expression*
THEN *expression*, …
[**ELSE** *expression*]
END

If the expression following the CASE statement is equal to the expression following the WHEN statement, then the expression following the THEN statement is returned. Otherwise the expression following the ELSE statement is returned, if it exists.

For example, the following code uses a case expression as the second clause in a SELECT statement.

```
SELECT ID,
   ( CASE Name
      WHEN 'Tee Shirt' then 'Shirt'
      WHEN 'Sweatshirt' then 'Shirt'
      WHEN 'Baseball Cap' then 'Hat'
      ELSE 'Unknown'
   END ) as Type
FROM Products
```

An alternative syntax is as follows:

CASE
WHEN *search-condition*
THEN *expression*, …
[**ELSE** *expression*]
END

If the search-condition following the WHEN statement is satisfied, the expression following the THEN statement is returned. Otherwise the expression following the ELSE statement is returned, if it exists.

For example, the following statement uses a case expression as the third clause of a SELECT statement to associate a string with a search-condition.

```
SELECT ID, Name,
   ( CASE
      WHEN Name='Tee Shirt' then 'Sale'
      WHEN Quantity >= 50  then 'Big Sale'
      ELSE 'Regular price'
   END ) as Type
FROM Products
```

NULLIF function for abbreviated CASE expressions

The NULLIF function provides a way to write some CASE statements in short form. The syntax for NULLIF is as follows:

NULLIF (*expression-1*, *expression-2*)

NULLIF compares the values of the two expressions. If the first expression equals the second expression, NULLIF returns NULL. If the first expression does not equal the second expression, NULLIF returns the first expression.

CASE statement is different from CASE expression
Do not confuse the syntax of the CASE expression with that of the CASE statement. For information on the CASE statement, see “CASE statement” on page 359.

Compatibility of expressions

Default interpretation of delimited strings

SQL Anywhere employs the SQL/2003 convention, that strings enclosed in apostrophes are constant expressions, and strings enclosed in quotation marks (double quotes) are delimited identifiers (names for database objects).

The quoted_identifier option

SQL Anywhere provides a quoted_identifier option that allows the interpretation of delimited strings to be changed. By default, the quoted_identifier option is set to On in SQL Anywhere. See "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

You cannot use SQL reserved words as identifiers if the quoted_identifier option is off.

For a complete list of reserved words, see "Reserved words" on page 4.

Setting the option

The following statement in SQL Anywhere changes the setting of the quoted_identifier option to On:

```
SET quoted_identifier On
```

The following statement in SQL Anywhere changes the setting of the quoted_identifier option to Off:

```
SET quoted_identifier Off
```

Compatible interpretation of delimited strings

You can choose to use either the SQL/2003 or the default Transact-SQL convention in SQL Anywhere as long as the quoted_identifier option is set to the same value in each DBMS.

Examples

If you choose to operate with the quoted_identifier option On (the default SQL Anywhere setting), then the following statements involving the SQL keyword **user** are valid for both DBMSs.

```
CREATE TABLE "user" (
   col1 char(5)
) ;
INSERT "user" ( col1 )
VALUES ( 'abcde' ) ;
```

If you choose to operate with the quoted_identifier option off then the following statements are valid for both DBMSs.

```
SELECT *
FROM Employees
WHERE Surname = "Chin"
```


Search conditions

A search condition is the criteria specified for a WHERE clause, a HAVING clause, a CHECK clause, an ON phrase in a join, or an IF expression.

Syntax

search-condition:
expression compare expression
| *expression compare* { [**ANY** | **SOME**] | **ALL** } (*subquery*)
| *expression* **IS** [**NOT**] **NULL**
| *expression* [**NOT**] **BETWEEN** *expression* **AND** *expression*
| *expression* [**NOT**] **LIKE** *expression* [**ESCAPE** *expression*]
| *expression* [**NOT**] **IN** **(** { *expression*
| *subquery*
| *value-expr1* **,** ... } **)**
| **EXISTS (** *subquery* **)**
| **NOT** *condition*
| *search-condition* **AND** *search-condition*
| *search-condition* **OR** *search-condition*
| **(** *search-condition* **)**
| **(** *search-condition* **,** *estimate* **)**
| *search-condition* **IS** [**NOT**] { **TRUE** | **FALSE** | **UNKNOWN** }
| *trigger-operation*

Parameters

compare:
= | **>** | **<** | **>=** | **<=** | **<>** | **!=** | **!<** | **!>**

trigger-operation:
INSERTING | **DELETING**
| **UPDATING** [**(** *column-name-string* **)**] | **UPDATE(** *column-name* **)**

Remarks

Search conditions are used to choose a subset of the rows from a table, or in a control statement such as an IF statement to determine control of flow.

In SQL, every condition evaluates as one of TRUE, FALSE, or UNKNOWN. This is called three-valued logic. The result of a comparison is UNKNOWN if either value being compared is the NULL value. For tables displaying how logical operators combine in three-valued logic, see "Three-valued logic" on page 27.

Rows satisfy a search condition if and only if the result of the condition is TRUE. Rows for which the condition is UNKNOWN or FALSE do not satisfy the search condition. For more information about NULL, see "NULL value" on page 43.

Subqueries form an important class of expression that is used in many search conditions. For information about using subqueries in search conditions, see "Subqueries in search conditions" on page 21.

The different types of search condition are discussed in the following sections.

Permissions

Must be connected to the database.

Side effects

None.

See also

- "Expressions" on page 15

Subqueries in search conditions

Subqueries that return exactly one column and either zero or one row can be used in any SQL statement wherever a column name could be used, including in the middle of an expression.

For example, expressions can be compared to subqueries in comparison conditions (see "Comparison operators" on page 11) as long as the subquery does not return more than one row. If the subquery (which must have exactly one column) returns one row, then the value of that row is compared to the expression. If a subquery returns no rows, the value of the subquery is NULL.

Subqueries that return exactly one column and any number of rows can be used in IN, ANY, ALL, and SOME search conditions. Subqueries that return any number of columns and rows can be used in EXISTS search conditions. These search conditions are discussed in the following sections.

ALL, ANY, and SOME search conditions

ANY search condition

The syntax for an ANY search condition is:

expression comparison-operator **ANY (** *subquery* **)**

where *comparison-operator* is one of <=, =, <, >, >=, <>, !<, !>, or !=.

The keyword SOME can be used instead of ANY.

With the ANY search condition, if the subquery result set is the empty set, the search condition evaluates to FALSE. Otherwise, the search condition evaluates to TRUE, FALSE, or UKNOWN, depending on the value of *expression*, and the result set returned by the subquery, as follows:

If the expression value is..	and the result set returned by the subquery contains at least one NULL, then..	or the result set returned by the subquery contains no NULLs, then..
NULL	UNKNOWN	UNKNOWN
not NULL	If there exists at least one value in the subquery result set for which the comparison with the expression value is TRUE, then the search condition evaluates to TRUE. Otherwise, the search condition evaluates to UNKNOWN.	If there exists at least one value in the subquery result set for which the comparison with the expression value is TRUE, then the search condition evaluates to TRUE. Otherwise, the search condition evaluates to FALSE.

For example, an ANY search condition with an equality operator,

expression = **ANY (** *subquery* **)**

evaluates to TRUE if *expression* is equal to any of the values in the result of the subquery, and FALSE if the value of the expression is not NULL, does not equal any of the values in the result of the subquery, and the result set doesn't contain NULLs.

> **Note**
> The usage of =ANY is equivalent to using the IN keyword.

ALL search condition

The syntax for an ALL search condition is:

expression comparison-operator **ALL (** *subquery* **)**

where *comparison-operator* is one of <=, =, <, >, >=, <>, !<, !>, or !=.

With the ALL search condition, if the value of subquery result set is the empty set, the search condition evaluates to TRUE. Otherwise, the search condition evaluates to TRUE, FALSE, or UKNOWN, depending on the value of *expression*, and the result set returned by the subquery, as follows:

If the expression value is..	and the result set returned by the subquery contains at least one NULL, then..	or the result set returned by the subquery contains no NULLs, then..
NULL	UNKNOWN	UNKNOWN
not NULL	If there exists at least one value in the subquery result set for which the comparison with the expression value is FALSE, then the search condition evaluates to FALSE. Otherwise, the search condition evaluates to UNKNOWN.	If there exists at least one value in the subquery result set for which the comparison with the expression value is FALSE, then the search condition evaluates to FALSE. Otherwise, the search condition evaluates to TRUE.

BETWEEN search condition

The syntax for the BETWEEN search condition is as follows:

expr [**NOT**] **BETWEEN** *start-expr* **AND** *end-expr*

The BETWEEN search condition can evaluate as TRUE, FALSE, or UNKNOWN. Without the NOT keyword, the search condition evaluates as TRUE if *expr* is between *start-expr* and *end-expr*. The NOT keyword reverses the meaning of the search condition but leaves UNKNOWN unchanged.

The BETWEEN search condition is equivalent to a combination of two inequalities:

[**NOT**] (*expr* >= *start-expr* **AND** *expr* <= *end-expr*)

LIKE search condition

The syntax for the LIKE search condition is as follows:

expression [**NOT**] **LIKE** *pattern* [**ESCAPE** *escape-expression*]

The LIKE search condition can evaluate as TRUE, FALSE, or UNKNOWN.

Without the NOT keyword, the search condition evaluates as TRUE if *expression* matches the *pattern*. If either *expression* or *pattern* is the NULL value, this search condition is UNKNOWN. The NOT keyword reverses the meaning of the search condition, but leaves UNKNOWN unchanged.

The pattern may contain any number of wildcards. The wildcards are:

Wildcard	Matches
_ (underscore)	Any one character, for example, a_
% (percent)	Any string of zero or more characters, for example bl%
[]	Any single character in the specified range or set, for example T[oi]m
[^]	Any single character *not* in the specified range or set, 'M[^c]%'

All other characters must match exactly.

For example, the search condition

```
... name LIKE 'a%b_'
```

is TRUE for any row where name starts with the letter a and has the letter b as its second last character.

If an *escape-expression* is specified, it must evaluate to a single character. The character can precede a percent, an underscore, a left square bracket, or another escape character in the *pattern* to prevent the special character from having its special meaning. When escaped in this manner, a percent will match a percent, and an underscore will match an underscore.

All patterns of length 126 characters or less are supported. Patterns of length greater than 254 characters are not supported. Some patterns of length between 127 and 254 characters are supported, depending on the contents of the pattern.

Searching for one of a set of characters

A set of characters to look for is specified by listing the characters inside square brackets. For example, the following search condition finds the strings *smith* and *smyth*:

```
LIKE 'sm[iy]th'
```

Searching for one of a range of characters

A range of characters to look for is specified by giving the ends of the range inside square brackets, separated by a hyphen. For example, the following search condition finds the strings bough and rough, but not tough:

```
LIKE '[a-r]ough'
```


The range of characters [a-z] is interpreted as "greater than or equal to a, and less than or equal to z", where the greater than and less than operations are carried out within the collation of the database. For information on ordering of characters within a collation, see "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*].

The lower end of the range must precede the higher end of the range. For example, a LIKE search condition containing the expression [z-a] returns no rows because no character matches the [z-a] range.

Unless the database is created as case sensitive, the range of characters is case insensitive. For example, the following search condition finds the strings Bough, rough, and TOUGH:

```
LIKE '[a-z]ough'
```

If the database is created as a case-sensitive database, the search condition is case sensitive also. To perform a case insensitive search in a case sensitive database, you must include upper and lower characters. For example, the following search condition finds the strings Bough, rough, and TOUGH:

```
LIKE '[a-zA-Z][oO][uU][gG][hH]'
```

Combining searches for ranges and sets

You can combine ranges and sets within a square bracket. For example, the following search condition finds the strings bough, rough, and tough:

```
... LIKE '[a-rt]ough'
```

The bracket [a-rt] is interpreted as "exactly one character that is either in the range a to r inclusive, or is t".

Searching for one character not in a range

The caret character (^) is used to specify a range of characters that is excluded from a search. For example, the following search condition finds the string tough, but not the strings rough, or bough:

```
... LIKE '[^a-r]ough'
```

The caret negates the entire rest of the contents of the brackets. For example, the bracket [^a-rt] is interpreted as "exactly one character that is not in the range a to r inclusive, and is not t".

Special cases of ranges and sets

Any single character in square brackets means that character. For example, [a] matches just the character a. [^] matches just the caret character, [%] matches just the percent character (the percent character does not act as a wildcard in this context), and [_] matches just the underscore character. Also, [[] matches just the character [.

Other special cases are as follows:

- The expression [a-] matches either of the characters a or -.
- The expression [] is never matched and always returns no rows.
- The expressions [or [abp-q are ill-formed expressions, and give syntax errors.
- You cannot use wildcards inside square brackets. The expression [a%b] finds one of a, %, or b.

- You cannot use the caret character to negate ranges except as the first character in the bracket. The expression [a^b] finds one of a, ^, or b.

Search patterns with trailing blanks

When your search pattern includes trailing blanks, SQL Anywhere matches the pattern only to values that contain blanks—it does not blank-pad strings. For example, the search patterns '90 ', '90[ ]' and '90_' match the value '90 ', but do not match the value '90', even if the value being tested is in a char or varchar column that is three or more characters in width.

Blank padded databases

A LIKE pattern in a LIKE predicate is a pattern-match representation whose semantics do not change if the database is blank-padded or not. Matching an expression to a LIKE pattern involves a character-by-character match of a value to the LIKE pattern, in a left-to-right fashion. No additional blank padding is performed on the value or expression during the evaluation. Therefore, the expression 'a' would match LIKE pattern 'a', but would not match LIKE patterns 'a ' (a, with a space after it), or 'a_'.

Standards and compatibility

- The ESCAPE clause is supported by SQL Anywhere only.

IN search condition

The syntax for the IN search condition is as follows:

expression [**NOT**] **IN** { **(** *subquery* **)** | **(** *expression2* **)** | **(** *value-expr*, … **)** }

An IN search condition, without the NOT keyword, evaluates according to the following rules:

- TRUE if *expression* is not NULL and equals at least one of the values.
- UNKNOWN if *expression* is NULL and the values list is not empty, or if at least one of the values is NULL and *expression* does not equal any of the other values.
- FALSE if *expression* is NULL and *subquery* returns no values; or if *expression* is not NULL, none of the values are NULL, and *expression* does not equal any of the values.

The NOT keyword interchanges TRUE and FALSE.

The search condition *expression* **IN (** *values* **)** is equivalent to *expression* = **ANY (** *values* **)**.

The search condition *expression* **NOT IN (** *values* **)** is equivalent to *expression* <> **ALL (** *values* **)**.

The *value-expr* arguments are expressions that take on a single value, which may be a string, a number, a date, or any other SQL data type.

EXISTS search condition

The syntax for the EXISTS search condition is as follows:

EXISTS(*subquery* **)**

The EXISTS search condition is TRUE if the subquery result contains at least one row, and FALSE if the subquery result does not contain any rows. The EXISTS search condition cannot be UNKNOWN.

IS NULL and IS NOT NULL search conditions

The syntax for the IS NULL search conditions is as follows:

expression **IS** [**NOT**] **NULL**

Without the NOT keyword, the IS NULL search condition is TRUE if the expression is the NULL value, and FALSE otherwise. The NOT keyword reverses the meaning of the search condition.

Truth value search conditions

The syntax for truth-value search conditions is as follows:

IS [**NOT**] *truth-value*

Without the NOT keyword, the search condition is TRUE if the *condition* evaluates to the supplied *truth-value*, which must be one of TRUE, FALSE, or UNKNOWN. Otherwise, the value is FALSE. The NOT keyword reverses the meaning of the search condition, but leaves UNKNOWN unchanged.

Standards and compatibility

- Vendor extension.

Trigger operation conditions

The syntax for trigger operation conditions is as follows:

trigger-operation:
INSERTING | **DELETING**
| **UPDATING** [**(** *column-name-string* **)**] | **UPDATE(** *column-name* **)**

Trigger-operation conditions can be used only in triggers, to carry out actions depending on the kind of action that caused the trigger to fire.

The argument for UPDATING is a quoted string (for example, `UPDATING( 'mycolumn' )`). The argument for UPDATE is an identifier (for example, `UPDATE( mycolumn )`). The two versions are interoperable, and are included for compatibility with SQL dialects of other vendors' DBMS.

If you supply an UPDATING or UPDATE function, you must also supply a REFERENCING clause in the CREATE TRIGGER statement to avoid syntax errors.

Example

The following trigger displays a message showing which action caused the trigger to fire.


```
CREATE TRIGGER tr BEFORE INSERT, UPDATE, DELETE
ON sample_table
REFERENCING OLD AS t1old
FOR EACH ROW
BEGIN
    DECLARE msg varchar(255);

    SET msg = 'This trigger was fired by an ';
    IF INSERTING THEN
        SET msg = msg || 'insert'
    ELSEIF DELETING THEN
        set msg = msg || 'delete'
    ELSEIF UPDATING THEN
        set msg = msg || 'update'
    END IF;
    MESSAGE msg TO CLIENT
END
```

Three-valued logic

The following tables display how the AND, OR, NOT, and IS logical operators of SQL work in three-valued logic.

AND operator

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR operator

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT operator

TRUE	FALSE	UNKNOWN
FALSE	TRUE	UNKNOWN

IS operator

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE

IS	TRUE	FALSE	UNKNOWN
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

Explicit selectivity estimates

SQL Anywhere uses statistical information to determine the most efficient strategy for executing each statement. SQL Anywhere automatically gathers and updates these statistics. These statistics are stored permanently in the database in the system table ISYSCOLSTAT. Statistics gathered while processing one statement are available when searching for efficient ways to execute subsequent statements.

Occasionally, the statistics may become inaccurate or relevant statistics may be unavailable. This condition is most likely to arise when few queries have been executed since a large amount of data was added, updated, or deleted. In this situation, you may want to execute CREATE STATISTICS.

If there are problems with a particular execution plan, you can use optimizer hints to require that a particular index be used. For more information, see "FROM clause" on page 535.

In unusual circumstances, however, these measures may prove ineffective. In such cases, you can sometimes improve performance by supplying explicit selectivity estimates.

For each table in a potential execution plan, the optimizer must estimate the number of rows that will be part of the result set. If you know that a condition has a success rate that differs from the optimizer's estimate, you can explicitly supply a user estimate in the search condition.

The estimate is a percentage. It can be a positive integer or can contain fractional values.

Caution
Whenever possible, avoid supplying explicit estimates in statements that are to be used on an ongoing basis. Should the data change, the explicit estimate may become inaccurate and may force the optimizer to select poor plans. If you do use explicit selectivity estimates, ensure that the number is accurate. Do not, for example, supply values of 0% or 100% to force the use of an index.

You can disable user estimates by setting the database option user_estimates to Off. The default value for user_estimates is Override-Magic, which means that user-supplied selectivity estimates are used only when the optimizer would use a MAGIC (default) selectivity value for the condition. The optimizer uses MAGIC values as a last resort when it is unable to accurately predict the selectivity of a predicate.

For more information about disabling user-defined selectivity estimates, see "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about statistics, see "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*].

Examples

- The following query provides an estimate that one percent of the ShipDate values will be later than 2001/06/30:

```
SELECT  ShipDate
   FROM  SalesOrderItems
WHERE ( ShipDate > '2001/06/30', 1 )
ORDER BY ShipDate DESC
```

- The following query estimates that half a percent of the rows will satisfy the condition:

```
SELECT *
   FROM Customers c, SalesOrders o
WHERE (c.ID = o.CustomerID, 0.5)
```

Fractional values enable more accurate user estimates for joins, particularly for large tables.

Special values

Special values can be used in expressions, and as column defaults when creating tables.

While some special values can be queried, some can only be used as default values for columns. For example, **user**, **last user**, **timestamp** and **UTC timestamp** can only be used as default values.

CURRENT DATABASE special value

CURRENT DATABASE returns the name of the current database.

Data type

STRING

See also

- "Expressions" on page 15

CURRENT DATE special value

CURRENT DATE returns the current year, month, and day.

Data type

DATE

See also

- "Expressions" on page 15
- "TIME data type" on page 72

CURRENT PUBLISHER special value

CURRENT PUBLISHER returns a string that contains the publisher user ID of the database for SQL Remote replications.

Data type

STRING

Remarks

CURRENT PUBLISHER can be used as a default value in columns with character data types.

See also

- "Expressions" on page 15
- "SQL Remote Installation Design" [*SQL Remote*]

CURRENT TIME special value

The current hour, minute, second and fraction of a second.

Data type

TIME

Remarks

The fraction of a second is stored to 6 decimal places. The accuracy of the current time is limited by the accuracy of the system clock.

See also

- "Expressions" on page 15
- "TIME data type" on page 72

CURRENT TIMESTAMP special value

CURRENT TIMESTAMP combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value containing the year, month, day, hour, minute, second and fraction of a second. The fraction of a second is stored to 3 decimal places. The accuracy is limited by the accuracy of the system clock.

Unlike DEFAULT TIMESTAMP, columns declared with DEFAULT CURRENT TIMESTAMP do not necessarily contain unique values. If uniqueness is required, consider using DEFAULT TIMESTAMP instead.

The information CURRENT TIMESTAMP returns is equivalent to the information returned by the GETDATE and NOW functions.

CURRENT_TIMESTAMP is equivalent to CURRENT TIMESTAMP.

Note
The main difference between DEFAULT CURRENT TIMESTAMP and DEFAULT TIMESTAMP is that DEFAULT CURRENT TIMESTAMP is set only at INSERT, while DEFAULT TIMESTAMP is set at both INSERT and UPDATE.

Data type

TIMESTAMP

See also

- "CURRENT TIME special value" on page 31
- "TIMESTAMP special value" on page 33
- "Expressions" on page 15
- "TIMESTAMP data type" on page 73
- "GETDATE function [Date and time]" on page 169
- "NOW function [Date and time]" on page 210

CURRENT USER special value

CURRENT USER returns a string that contains the user ID of the current connection.

Data type

STRING

Remarks

CURRENT USER can be used as a default value in columns with character data types.

On UPDATE, columns with a default value of CURRENT USER are not changed. CURRENT_USER is equivalent to CURRENT USER.

See also

- "Expressions" on page 15

CURRENT UTC TIMESTAMP special value

CURRENT UTC TIMESTAMP combines CURRENT DATE and CURRENT TIME, adjusted by the server's time zone adjustment value, to form a Coordinated Universal Time (UTC) TIMESTAMP value containing the year, month, day, hour, minute, second and fraction of a second. This feature allows data to be entered with a consistent time reference, regardless of the time zone in which the data was entered.

Data type

TIMESTAMP

See also

- "TIMESTAMP data type" on page 73
- "UTC TIMESTAMP special value" on page 35
- "CURRENT TIMESTAMP special value" on page 31
- "truncate_timestamp_values option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]

LAST USER special value

LAST USER is the name of the user who last modified the row.

Data type

String.

Remarks

LAST USER can be used as a default value in columns with character data types.

On INSERT, this constant has the same effect as CURRENT USER. On UPDATE, if a column with a default value of LAST USER is not explicitly modified, it is changed to the name of the current user.

When combined with the DEFAULT TIMESTAMP, a default value of LAST USER can be used to record (in separate columns) both the user and the date and time a row was last changed.

See also

- "CURRENT USER special value" on page 32
- "CURRENT TIMESTAMP special value" on page 31
- "CREATE TABLE statement" on page 450

SQLCODE special value

SQLCODE is the current SQLCODE value.

Data type

String.

Remarks

The SQLCODE value is set after each statement. You can check the SQLCODE to see whether or not the statement succeeded.

See also

- "Expressions" on page 15
- SQL Anywhere 10 - Error Messages [*SQL Anywhere 10 - Error Messages*].

SQLSTATE special value

SQLSTATE is the current SQLSTATE value

Data type

STRING

Remarks

The SQLSTATE value is set after each statement. You can check the SQLSTATE to see whether or not the statement succeeded.

See also

- "Expressions" on page 15
- SQL Anywhere 10 - Error Messages [*SQL Anywhere 10 - Error Messages*]

TIMESTAMP special value

TIMESTAMP indicates when each row in the table was last modified. When a column is declared with DEFAULT TIMESTAMP, a default value is provided for inserts, and the value is updated with the current date and time whenever the row is updated.

Data type

TIMESTAMP

Remarks

Columns declared with DEFAULT TIMESTAMP contain unique values so that applications can detect near-simultaneous updates to the same row. If the current timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option.

You can automatically truncate timestamp values in SQL Anywhere based on the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values.

The global variable @@dbts returns a TIMESTAMP value representing the last value generated for a column using DEFAULT TIMESTAMP.

Note
The main difference between DEFAULT TIMESTAMP and DEFAULT CURRENT TIMESTAMP is that DEFAULT CURRENT TIMESTAMP is set only at INSERT, while DEFAULT TIMESTAMP is set at both INSERT and UPDATE.

See also

- "TIMESTAMP data type" on page 73
- "CURRENT TIMESTAMP special value" on page 31
- "CURRENT UTC TIMESTAMP special value" on page 32
- "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]
- "truncate_timestamp_values option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]

USER special value

USER returns a string that contains the user ID of the current connection.

Data type

STRING

USER can be used as a default value in columns with character data types.

Remarks

On UPDATE, columns with a default value of USER are not changed.

See also

- "Expressions" on page 15
- "CURRENT USER special value" on page 32
- "LAST USER special value" on page 32

UTC TIMESTAMP special value

UTC TIMESTAMP indicates the Coordinated Universal (UTC) time when each row in the table was last modified.

When a column is declared with DEFAULT UTC TIMESTAMP, a default value is provided for inserts, and the value is updated with the current UTC date and time whenever the row is updated.

Data type

TIMESTAMP

Remarks

Columns declared with DEFAULT UTC TIMESTAMP contain unique values so that applications can detect near-simultaneous updates to the same row. If the current UTC timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option.

You can automatically truncate UTC timestamp values in SQL Anywhere with the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values.

Note
The main difference between DEFAULT UTC TIMESTAMP and DEFAULT CURRENT UTC TIMESTAMP is that DEFAULT CURRENT UTC TIMESTAMP is set only at INSERT, while DEFAULT UTC TIMESTAMP is set at both INSERT and UPDATE.

See also

- "TIMESTAMP data type" on page 73
- "CURRENT UTC TIMESTAMP special value" on page 32
- "TIMESTAMP special value" on page 33
- "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]
- "truncate_timestamp_values option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]

Variables

SQL Anywhere supports three levels of variables:

- **Local variables** These are defined inside a compound statement in a procedure or batch using the DECLARE statement. They exist only inside the compound statement.
- **Connection-level variables** These are defined with a CREATE VARIABLE statement. They belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement.
- **Global variables** These are system-supplied variables that have system-supplied values. All global variables have names beginning with two @ signs. For example, the global variable **@@version** has a value that is the current version number of the database server. Users cannot define global variables.

Local and connection-level variables are declared by the user, and can be used in procedures or in batches of SQL statements to hold information. Global variables are system-supplied variables that provide system-supplied values.

See also

- "TIMESTAMP data type" on page 73
- "CREATE VARIABLE statement" on page 469

Local variables

SQL Anywhere supports local variables. Local variables are declared using the DECLARE statement, which can be used only within a compound statement (that is, bracketed by the BEGIN and END keywords). Only one variable can be declared for each DECLARE statement in SQL Anywhere.

If the DECLARE is executed within a compound statement, the scope is limited to the compound statement.

The variable is initially set as NULL. The value of the variable can be set using the SET statement, or can be assigned using a SELECT statement with an INTO clause.

The syntax of the DECLARE statement is as follows:

```
DECLARE variable-name data-type
```

Local variables can be passed as arguments to procedures, as long as the procedure is called from within the compound statement.

Examples

- The following batch illustrates the use of local variables.

```
BEGIN
   DECLARE local_var INT;
   SET local_var = 10;
   MESSAGE 'local_var = ', local_var TO CLIENT;
END
```


Running this batch from Interactive SQL gives the message `local_var = 10` in the Interactive SQL Messages tab.

- The variable local_var does not exist outside the compound statement in which it is declared. The following batch is invalid, and gives a `column not found` error.

```
-- This batch is invalid.
BEGIN
   DECLARE local_var INT;
   SET local_var = 10;
END;
MESSAGE 'local_var = ', local_var TO CLIENT;
```

- The following example illustrates the use of SELECT with an INTO clause to set the value of a local variable:

```
BEGIN
   DECLARE local_var INT;
   SELECT 10 INTO local_var;
   MESSAGE 'local_var = ', local_var TO CLIENT;
END
```

Running this batch from Interactive SQL gives the message `local_var = 10` on the Server Messages window.

For more information on batches and local variable scope, see "Variables in Transact-SQL procedures" [*SQL Anywhere Server - SQL Usage*].

Connection-level variables

Connection-level variables are declared with the CREATE VARIABLE statement. Connection-level variables can be passed as parameters to procedures.

The syntax for the CREATE VARIABLE statement is as follows:

```
CREATE VARIABLE variable-name data-type
```

When a variable is created, it is initially set to NULL. The value of connection-level variables can be set in the same way as local variables, using the SET statement or using a SELECT statement with an INTO clause.

Connection-level variables exist until the connection is terminated, or until the variable is explicitly dropped using the DROP VARIABLE statement. The following statement drops the variable con_var:

```
DROP VARIABLE con_var
```

Example

- The following batch of SQL statements illustrates the use of connection-level variables.

```
CREATE VARIABLE con_var INT;
SET con_var = 10;
MESSAGE 'con_var = ', con_var TO CLIENT;
```

Running this batch from Interactive SQL gives the message `con_var = 10` on the Server Messages window.

Global variables

Global variables have values set by the database server. For example, the global variable @@version has a value that is the current version number of the database server.

Global variables are distinguished from local and connection-level variables by having two @ signs preceding their names. For example, @@error and @@rowcount are global variables. Users cannot create global variables, and cannot update the values of global variables directly.

Some global variables, such as @@identity, hold connection-specific information, and so have connection-specific values. Other variables, such as @@connections, have values that are common to all connections.

Global variable and special constants

The special constants (for example, CURRENT DATE, CURRENT TIME, USER, and SQLSTATE) are similar to global variables.

The following statement retrieves a value of the version global variable.

```
SELECT @@version;
```

In procedures and triggers, global variables can be selected into a variable list. The following procedure returns the server version number in the *ver* parameter.

```
CREATE PROCEDURE VersionProc (OUT ver VARCHAR(100))
 BEGIN
   SELECT @@version
   INTO ver;
 END;
```

In Embedded SQL, global variables can be selected into a host variable list.

List of global variables

The following table lists the global variables available in SQL Anywhere. Some global variables are supplied for compatibility with Transact-SQL, and return a fixed value of either 0, 1, or NULL, as noted.

Variable name	Meaning
@@char_convert	0 (Provided for compatibility with Transact-SQL.)
@@client_csid	–1 (Provided for compatibility with Transact-SQL.)
@@client_csname	NULL (Provided for compatibility with Transact-SQL.)
@@connections	The number of logins since the server was last started
@@cpu_busy	0 (Provided for compatibility with Transact-SQL.)
@@dbts	A value of type TIMESTAMP representing the last generated value used for all columns defined with DEFAULT TIMESTAMP.

Variable name	Meaning
@@error	Commonly used to check the error status (succeeded or failed) of the most recently executed statement. It contains 0 if the previous transaction succeeded; otherwise, it contains the last error number generated by the system. A statement such as `if @@error != 0 return` causes an exit if an error occurs. Every statement resets @@error, including PRINT statements or IF tests, so the status check must immediately follow the statement whose success is in question.
@@fetch_status	Contains status information resulting from the last fetch statement. This feature is the same as @@sqlstatus, except that it returns different values. It is for Microsoft SQL Server compatibility. @@fetch_status may contain the following values: ♦ 0 The fetch statement completed successfully. ♦ -1 The fetch statement resulted in an error. ♦ -2 There is no more data in the result set.
@@identity	Last value inserted into any IDENTITY or DEFAULT AUTOINCREMENT column by an INSERT or SELECT INTO statement. See "@@identity global variable" on page 41.
@@idle	0 (Provided for compatibility with Transact-SQL.)
@@io_busy	0 (Provided for compatibility with Transact-SQL.)
@@isolation	Current isolation level of the connection. @@isolation takes the value of the active level.
@@langid	Unique language ID for the language in use by the current connection.
@@language	Name of the language in use by the connection.
@@max_connections	For the personal server, the maximum number of simultaneous connections that can be made to the server, which is 10. For the network server, the maximum number of active clients (not database connections, as each client can support multiple connections).
@@maxcharlen	Maximum length, in bytes, of a character in the CHAR character set.
@@ncharsize	Maximum length, in bytes, of a character in the NCHAR character set.
@@nestlevel	-1 (Provided for compatibility with Transact-SQL.)
@@pack_received	0 (Provided for compatibility with Transact-SQL.)
@@pack_sent	0 (Provided for compatibility with Transact-SQL.)
@@packet_errors	0 (Provided for compatibility with Transact-SQL.)
@@procid	Stored procedure ID of the currently executing procedure.

Variable name	Meaning
@@rowcount	Number of rows affected by the last statement. The value of @@rowcount should be checked immediately after the statement. Inserts, updates, and deletes set @@rowcount to the number of rows affected. With cursors, @@rowcount represents the cumulative number of rows returned from the cursor result set to the client, up to the last fetch request. The @@rowcount is not reset to zero by any statement which does not affect rows, such as an IF statement.
@@servername	Name of the current database server.
@@spid	The connection handle for the current connection. This is the same value as that displayed by the sa_conn_info procedure.
@@sqlstatus	Contains status information resulting from the last fetch statement. @@sqlstatus may contain the following values: ♦ 0 The fetch statement completed successfully. ♦ 1 The fetch statement resulted in an error. ♦ 2 There is no more data in the result set.
@@textsize	Current value of the SET TEXTSIZE option, which specifies the maximum length, in bytes, of text or image data to be returned with a select statement. The default setting is 32765, which is the largest bytestring that can be returned using READTEXT. The value can be set using the SET statement.
@@thresh_hysteresis	0 (Provided for compatibility with Transact-SQL.)
@@timeticks	0 (Provided for compatibility with Transact-SQL.)
@@total_errors	0 (Provided for compatibility with Transact-SQL.)
@@total_read	0 (Provided for compatibility with Transact-SQL.)
@@total_write	0 (Provided for compatibility with Transact-SQL.)
@@tranchained	Current transaction mode; 0 for unchained or 1 for chained.
@@trancount	Nesting level of transactions. Each BEGIN TRANSACTION in a batch increments the transaction count.
@@transtate	-1 (Provided for compatibility with Transact-SQL.)
@@version	Version number of the current version of SQL Anywhere.

@@identity global variable

The @@identity variable holds the most recent value inserted into an IDENTITY column or a DEFAULT AUTOINCREMENT column, or zero if the most recent insert was into a table that had no such column.

The value of @@identity is connection specific. It is reset each time a row is inserted into a table. If a statement inserts multiple rows, @@identity reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, @@ identity is set to 0.

The value of @@identity is not affected by the failure of an INSERT or SELECT INTO statement, or the rollback of the transaction that contained it. @@identity retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.

@@identity and triggers

When an insert causes referential integrity actions or fires a trigger, @@identity behaves like a stack. For example, if an insert into a table T1 (with an identity or autoincrement column) fires a trigger that inserts a row into table T2 (also with an identity or autoincrement column), then the value returned to the application or procedure which carried out the insert is the value inserted into T1. Within the trigger, @@identity has the T1 value before the insert into T2 and the T2 value after. The trigger can copy the values to local variables if it needs to access both.

Comments

Comments are used to attach explanatory text to SQL statements or statement blocks. The database server does not execute comments.

Several comment indicators are available in SQL Anywhere.

- **-- (Double hyphen)** The database server ignores any remaining characters on the line. This is the SQL/2003 comment indicator.
- **// (Double slash)** The double slash has the same meaning as the double hyphen.
- **/* … */ (Slash-asterisk)** Any characters between the two comment markers are ignored. The two comment markers may be on the same or different lines. Comments indicated in this style can be nested. This style of commenting is also called C-style comments.
- **% (Percent sign)** The percent sign has the same meaning as the double hyphen, if the percent_as_comment option is set to On. It is recommended that % not be used as a comment indicator.

Examples

- The following example illustrates the use of double-hyphen comments:

```
CREATE FUNCTION fullname ( firstname CHAR(30),
        lastname CHAR(30))
RETURNS CHAR(61)
-- fullname concatenates the firstname and lastname
-- arguments with a single space between.
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN ( name );
END
```

- The following example illustrates the use of C-style comments:

```
/*
   Lists the names and employee IDs of employees
   who work in the sales department.
*/
CREATE VIEW SalesEmployees AS
SELECT EmployeeID, Surname, GivenName
FROM Employees
WHERE DepartmentID = 200
```


NULL value

The NULL value specifies a value that is unknown or not applicable.

Syntax

NULL

Remarks

NULL is a special value that is different from any valid value for any data type. However, the NULL value is a legal value in any data type. NULL is used to represent missing or inapplicable information. There are two separate and distinct cases where NULL is used:

Situation	Description
missing	The field does have a value, but that value is unknown.
inapplicable	The field does not apply for this particular row.

SQL allows columns to be created with the NOT NULL restriction. This means that those particular columns cannot contain NULL.

The NULL value introduces the concept of three valued logic to SQL. The NULL value compared using any comparison operator with any value (including the NULL value) is "UNKNOWN." The only search condition that returns TRUE is the IS NULL predicate. In SQL, rows are selected only if the search condition in the WHERE clause evaluates to TRUE; rows that evaluate to UNKNOWN or FALSE are not selected.

The IS [NOT] *truth-value* clause, where *truth-value* is one of TRUE, FALSE or UNKNOWN can be used to select rows where the NULL value is involved. See "Search conditions" on page 20 for a description of this clause.

In the following examples, the column Salary contains NULL.

Condition	Truth value	Selected?
Salary = NULL	UNKNOWN	NO
Salary <> NULL	UNKNOWN	NO
NOT (Salary = NULL)	UNKNOWN	NO
NOT (Salary <> NULL)	UNKNOWN	NO
Salary = 1000	UNKNOWN	NO
Salary IS NULL	TRUE	YES
Salary IS NOT NULL	FALSE	NO
Salary = *expression* IS UNKNOWN	TRUE	YES

The same rules apply when comparing columns from two different tables. Therefore, joining two tables together will not select rows where any of the columns compared contain the NULL value.

NULL also has an interesting property when used in numeric expressions. The result of *any* numeric expression involving the NULL value is NULL. This means that if NULL is added to a number, the result is NULL—not a number. If you want NULL to be treated as 0, you must use the **ISNULL(** *expression*, **0)** function (see "SQL Functions" on page 91).

Many common errors in formulating SQL queries are caused by the behavior of NULL. You will have to be careful to avoid these problem areas. See "Search conditions" on page 20 for a description of the effect of three-valued logic when combining search conditions.

Set operators and DISTINCT clause

In set operations (UNION, INTERSECT, EXCEPT), and in the DISTINCT operation, NULL is treated differently from in search conditions. Rows that contain NULL and are otherwise identical are treated as identical for the purposes of these operations.

For example, if a column called redundant contained NULL for every row in a table T1, then the following statement would return a single row:

```
SELECT DISTINCT redundant FROM T1
```

Permissions

Must be connected to the database.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.
- **Sybase** In some contexts, Adaptive Server Enterprise treats NULL as a value, whereas SQL Anywhere does not. For example, rows of a column c1 that are NULL are not included in the results of a query with the following WHERE clause in SQL Anywhere, as the condition has a value of UNKNOWN:

  ```
  WHERE NOT( C1 = NULL )
  ```

  In Adaptive Server Enterprise, the condition is evaluated as TRUE, and these rows are returned. You should use IS NULL rather than a comparison operator for compatibility.

  Unique indexes in SQL Anywhere can hold rows that hold NULL and are otherwise identical. Adaptive Server Enterprise does not permit such entries in unique indexes.

  If you use jConnect, the tds_empty_string_is_null option controls whether empty strings are returned as NULL strings or as a string containing one blank character.

  For more information, see "tds_empty_string_is_null option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- “Expressions” on page 15
- “Search conditions” on page 20

Example

- The following INSERT statement inserts a NULL into the date_returned column of the Borrowed_book table.

```
INSERT
INTO Borrowed_book
( date_borrowed, date_returned, book )
VALUES ( CURRENT DATE, NULL, '1234' )
```


CHAPTER 2

SQL Data Types

Contents

Character data types 48
Numeric data types 56
Money data types 64
Bit array data types 65
Date and time data types 67
Binary data types 74
Domains 78
Data type conversions 80
Java and SQL data type conversion 88

Character data types

Character data types are used to store strings of letters, numbers, and other symbols.

SQL Anywhere provides two classes of character data types and some domains defined using those types.

- **CHAR, VARCHAR, LONG VARCHAR** Character data stored in a single- or multibyte character set, often chosen to correspond most closely to the primary language or languages stored in the database.
- **NCHAR, NVARCHAR, LONG NVARCHAR** Character data stored in Unicode's UTF-8 encoding. All Unicode code points can be stored using these types, regardless of the primary language or languages stored in the database.
- **TEXT, UNIQUEIDENTIFIERSTR, XML** Domains based on other character data types.

Storage

All character data values are stored in the same manner. By default, values up to 128 bytes are stored in a single piece. Values longer than 128 bytes are stored with a 4-byte prefix kept locally on the database page and the full value stored in one or more other database pages. These default sizes are controlled by the INLINE and PREFIX clauses of the CREATE TABLE statement.

See also

- "CREATE TABLE statement" on page 450
- "string_rtruncation option [compatibility]" [*SQL Anywhere Server - Database Administration*]

CHAR data type

The CHAR data type stores character data, up to 32767 bytes.

Syntax

CHAR [**(** *max-length* [**CHAR** | **CHARACTER**] **)**]

Parameters

max-length The maximum length of the string. If byte-length semantics are used (CHAR or CHARACTER is not specified as part of the length), then the length is in bytes, and the length must be in the range 1 to 32767. If the length is not specified, then it is 1.

If character-length semantics are used (CHAR or CHARACTER is specified as part of the length), then the length is in characters, and you must specify *max-length*. When using character-length semantics, the length multiplied by the maximum length of a character in the database encoding must not exceed 32767 bytes. The following table shows the maximum lengths for the supported types of character sets:

Character set	Maximum length of CHAR
Single-byte character set	32767 bytes
Double-byte character set	16383 bytes

Character set	Maximum length of CHAR
UTF-8	8191 bytes

Remarks

Multibyte characters can be stored as CHAR, but the declared length refers to bytes, not characters, unless character-length semantics are used.

CHAR can also be specified as CHARACTER. Regardless of which syntax is used, the data type is described as CHAR.

CHAR is semantically equivalent to VARCHAR, although they are different types. In SQL Anywhere, CHAR is a variable-length type. In other relational database management systems, CHAR is a fixed-length type, and data is padded with blanks to *max-length* bytes of storage. SQL Anywhere does not blank-pad stored character data.

Using character-length semantics may impact what is returned when a client application performs a DESCRIBE on a column, depending on the interface used. For example, when an embedded SQL client performs a DESCRIBE on a column that was declared using byte-length semantics, the length returned is the byte length specified. Consequently, a CHAR(10) column is described as type DT_FIXCHAR with a length of 10 bytes. However, when an embedded SQL client performs a DESCRIBE on a column that was declared using character-length semantics, the length returned is the maximum byte length in the client's CHAR character set. For example, for an embedded SQL client using UTF-8 as the CHAR character set, a CHAR(10 CHAR) column is described as type DT_FIXCHAR with a length of 40 bytes (10 characters multiplied by the maximum of four bytes per character).

See also

- "VARCHAR data type" on page 53
- "LONG VARCHAR data type" on page 50
- "NCHAR data type" on page 50

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. Character-length semantics is a vendor extension.

LONG NVARCHAR data type

The LONG NVARCHAR data type stores Unicode character data of arbitrary length.

Syntax

LONG NVARCHAR

Remarks

The maximum size is 2 GB.

Characters are stored in UTF-8. Each character requires from one to four bytes. The maximum number of characters that can be stored in a LONG NVARCHAR is over 500 million and possibly over 2 billion, depending on the lengths of the characters stored.

When an embedded SQL client performs a DESCRIBE on a LONG NVARCHAR column, the data type returned is either DT_LONGVARCHAR or DT_LONGNVARCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

For ODBC, a LONG NVARCHAR expression is described as SQL_WLONGVARCHAR.

See also

- "NCHAR data type" on page 50
- "NVARCHAR data type" on page 52
- "LONG VARCHAR data type" on page 50

Standards and compatibility

- **SQL/2003** Vendor extension.

LONG VARCHAR data type

The LONG VARCHAR data type stores character data of arbitrary length.

Syntax

LONG VARCHAR

Remarks

The maximum size is 2 GB.

Multibyte characters can be stored as LONG VARCHAR, but the length is in bytes, not characters.

See also

- "CHAR data type" on page 48
- "VARCHAR data type" on page 53
- "LONG NVARCHAR data type" on page 49

Standards and compatibility

- **SQL/2003** Vendor extension.

NCHAR data type

The NCHAR data type stores Unicode character data, up to 8191 characters.

Syntax

NCHAR [**(** *max-length* **)**]

Parameters

max-length The maximum length of the string, in characters. The length must be in the range 1 to 8191. If the length is not specified, then it is 1.

Remarks

Characters are stored using UTF-8 encoding. The maximum number of bytes of storage required is four multiplied by *max-length*, although the actual storage required is usually much less.

NCHAR can also be specified as NATIONAL CHAR or NATIONAL CHARACTER. Regardless of which syntax is used, the data type is described as NCHAR.

When an embedded SQL client performs a DESCRIBE on an NCHAR column, the data type returned is either DT_FIXCHAR or DT_NFIXCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

Also, when an embedded SQL client performs a DESCRIBE on an NCHAR column, the length returned is the maximum byte length in the client's NCHAR character set. For example, for an embedded SQL client using the Western European character set cp1252 as the NCHAR character set, an NCHAR(10) column is described as type DT_NFIXCHAR of length 10 (10 characters multiplied by a maximum one byte per character). For an embedded SQL client using the Japanese character set cp932, the same column is described as type DT_NFIXCHAR of length 20 (10 characters multiplied by a maximum two bytes per character).

NCHAR is semantically equivalent to NVARCHAR, although they are different types. In SQL Anywhere, NCHAR is a variable-length type. In other relational database management systems, NCHAR is a fixed-length type, and data is padded with blanks to *max-length* characters of storage. SQL Anywhere does not blank-pad stored character data.

For ODBC, NCHAR is described as either SQL_WCHAR or SQL_WVARCHAR depending on the odbc_distingish_char_and_varchar option. See "odbc_distinguish_char_and_varchar option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "CHAR data type" on page 48
- "NVARCHAR data type" on page 52
- "LONG NVARCHAR data type" on page 49

Standards and compatibility

- **SQL/2003** Vendor extension.

NTEXT data type

The NTEXT data type stores Unicode character data of arbitrary length.

Syntax

NTEXT

Remarks

NTEXT is a domain, implemented as a LONG NVARCHAR.

See also

- "LONG NVARCHAR data type" on page 49
- "TEXT data type" on page 53

Standards and compatibility

- **SQL/2003** Vendor extension.

NVARCHAR data type

The NVARCHAR data type stores Unicode character data, up to 8191 characters.

Syntax

NVARCHAR [**(** *max-length* **)**]

Parameters

max-length The maximum length of the string, in characters. The length must be in the range 1 to 8191. If the length is not specified, then it is 1.

Remarks

Characters are stored in UTF-8 encoding. The maximum storage number of bytes required is four multiplied by *max-length*, although the actual storage required is usually much less.

NVARCHAR can also be specified as NCHAR VARYING, NATIONAL CHAR VARYING, or NATIONAL CHARACTER VARYING. Regardless of which syntax is used, the data type is described as NVARCHAR.

When an embedded SQL client performs a DESCRIBE on a NVARCHAR column, the data type returned is either DT_VARCHAR or DT_NVARCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

Also, when an embedded SQL client performs a DESCRIBE on an NVARCHAR column, the length returned is the maximum byte length in the client's NCHAR character set. For example, for an embedded SQL client using the Western European character set cp1252 as the NCHAR character set, an NVARCHAR(10) column is described as type DT_NVARCHAR of length 10 (10 characters multiplied by a maximum of one byte per character). For an embedded SQL client using the Japanese character set cp932, the same column is described as type DT_NVARCHAR of length 20 (10 characters multiplied by a maximum two bytes per character).

For ODBC, NVARCHAR is described as SQL_WCHAR or SQL_WVARCHAR, depending on the odbc_distingish_char_and_varchar option. See "odbc_distinguish_char_and_varchar option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "NCHAR data type" on page 50
- "LONG NVARCHAR data type" on page 49
- "VARCHAR data type" on page 53

Standards and compatibility

- **SQL/2003** Vendor extension.

TEXT data type

The TEXT data type stores character data of arbitrary length.

Syntax

TEXT

Remarks

TEXT is a domain, implemented as a LONG VARCHAR.

See also

- "LONG VARCHAR data type" on page 50
- "NTEXT data type" on page 51

Standards and compatibility

- **SQL/2003** Vendor extension.

UNIQUEIDENTIFIERSTR data type

The UNIQUEIDENTIFIERSTR data type is a domain implemented as CHAR(36).

Syntax

UNIQUEIDENTIFIERSTR

Remarks

Used for remote data access, when mapping Microsoft SQL Server uniqueidentifier columns.

See also

- "Data type conversions: Microsoft SQL Server" [*SQL Anywhere Server - SQL Usage*]
- "STRTOUUID function [String]" on page 261

Standards and compatibility

- **SQL/2003** Vendor extension.

VARCHAR data type

The VARCHAR data type stores character data, up to 32767 bytes.

Syntax

VARCHAR [**(** *max-length* [**CHAR** | **CHARACTER**] **)**]

Parameters

max-length The maximum length of the string. If byte-length semantics are used (CHAR or CHARACTER is *not* specified as part of the length), then the length is in bytes, and the length must be in the range of 1 to 32767. If the length is not specified, then it is 1.

If character-length semantics are used (CHAR or CHARACTER is specified as part of the length), then the length is in characters, and you must specify *max-length*. When using character-length semantics, the length multiplied by the maximum length of a character in the database encoding must not exceed 32767 bytes. The following table shows the maximum lengths for the supported types of character sets:

Character set	Maximum length of VARCHAR
Single-byte character set	32767 bytes
Double-byte character set	16383 bytes
UTF-8	8191 bytes

Remarks

Multibyte characters can be stored as VARCHAR, but the declared length refers to bytes, not characters.

VARCHAR can also be specified as CHAR VARYING or CHARACTER VARYING. Regardless of which syntax is used, the data type is described as VARCHAR.

Using character-length semantics may impact what is returned when a client application performs a DESCRIBE on a column, depending on the interface used. For example, when an embedded SQL client application performs a DESCRIBE on a column that was declared using byte-length semantics, the length returned is the byte length specified. Consequently, a VARCHAR(10) column is described as type DT_VARCHAR with a length of 10 bytes. However, when an embedded SQL client application performs a DESCRIBE on a column that was declared using character-length semantics, the length returned is the maximum byte length in the client's CHAR character set. For example, for a client that is using UTF-8 as the CHAR character set, a VARCHAR(10 CHAR) column is described as type DT_VARCHAR with a length of 40 bytes (10 characters multiplied by a maximum of four bytes per character).

See also

- "CHAR data type" on page 48
- "LONG VARCHAR data type" on page 50
- "NVARCHAR data type" on page 52

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. Character-length semantics is a vendor extension.

XML data type

The XML data type stores character data of arbitrary length, and is used to store XML documents.

Syntax

XML

Remarks

The maximum size is 2 GB.

Data of type XML is not quoted when generating element content from relational data.

You can cast between the XML data type and any other data type that can be cast to or from a string. Note that there is no checking that the string is well-formed when it is cast to XML.

For information about using the XML data type when generating XML elements, see "Storing XML documents in relational databases" [*SQL Anywhere Server - SQL Usage*].

When an embedded SQL client application performs a DESCRIBE on an XML column, it is described as LONG VARCHAR.

See also

- "Using XML in the Database" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

Numeric data types

The numeric data types are used for storing numerical data.

The NUMERIC and DECIMAL data types, and the various kinds of INTEGER data types, are sometimes called **exact** numeric data types, in contrast to the **approximate** numeric data types FLOAT, DOUBLE, and REAL.

The exact numeric data types are those for which precision and scale values can be specified, while approximate numeric data types are stored in a predefined manner. *Only exact numeric data is guaranteed accurate to the least significant digit specified after an arithmetic operation.*

Data type lengths and precision of less than one are not allowed.

Compatibility

Only the NUMERIC data type with scale = 0 can be used for the Transact-SQL identity column.

Be careful using default precision and scale settings for NUMERIC and DECIMAL data types, because these settings could be different in other database solutions. In SQL Anywhere, the default precision is 30 and the default scale is 6.

You should avoid default precision and scale settings for NUMERIC and DECIMAL data types, because these are different between SQL Anywhere and Adaptive Server Enterprise. In SQL Anywhere, the default precision is 30 and the default scale is 6. In Adaptive Server Enterprise, the default precision is 18 and the default scale is 0.

The FLOAT (*p*) data type is a synonym for REAL or DOUBLE, depending on the value of *p*. For SQL Anywhere, the cutoff is platform-dependent, but on all platforms the cutoff value is greater than 15.

For information about changing the defaults by setting database options, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

BIGINT data type

The BIGINT data type is used to store BIGINTs, which are integers requiring 8 bytes of storage.

Syntax

[**UNSIGNED**] **BIGINT**

Remarks

The BIGINT data type is an exact numeric data type: its accuracy is preserved after arithmetic operations.

A BIGINT value requires 8 bytes of storage.

The range for signed BIGINT values is -2^{63} to $2^{63} - 1$, or –9223372036854775808 to 9223372036854775807.

The range for unsigned BIGINT values is 0 to $2^{64} - 1$, or 0 to 18446744073709551615.

By default, the data type is signed.

See also

- "BIT data type" on page 57
- "INTEGER data type" on page 60
- "SMALLINT data type" on page 62
- "TINYINT data type" on page 63
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

BIT data type

The BIT data type is used to store a bit (0 or 1).

Syntax

BIT

Remarks

BIT is an integer type that can store the values 0 or 1.

By default, the BIT data type does not allow NULL.

See also

- "BIGINT data type" on page 56
- "INTEGER data type" on page 60
- "SMALLINT data type" on page 62
- "TINYINT data type" on page 63
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

DECIMAL data type

The DECIMAL data type is a decimal number with *precision* total digits and with *scale* digits after the decimal point.

Syntax

DECIMAL [**(** *precision* [, *scale*] **)**]

Parameters

precision An integer expression between 1 and 127, inclusive, that specifies the number of digits in the expression. The default setting is 30.

scale An integer expression between 0 and 127, inclusive, that specifies the number of digits after the decimal point. The scale value should always be less than, or equal to, the precision value. The default setting is 6.

The defaults can be changed by setting database options. For information, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

Remarks

The DECIMAL data type is an exact numeric data type; its accuracy is preserved to the least significant digit after arithmetic operations.

The storage required for a decimal number can be estimated as

```
2 + int( (before + 1)/2 ) + int( (after + 1)/2 )
```

The function int takes the integer portion of its argument, and before and after are the number of significant digits before and after the decimal point. The storage is based on the value being stored, not on the maximum precision and scale allowed in the column.

DECIMAL can also be specified as DEC. Regardless of which syntax is used, the data type is described as DECIMAL.

DECIMAL is semantically equivalent to NUMERIC.

See also

- "FLOAT data type" on page 59
- "REAL data type" on page 62
- "DOUBLE data type" on page 58
- "NUMERIC data type" on page 61
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

DOUBLE data type

The DOUBLE data type is used to store double-precision floating-point numbers.

Syntax

DOUBLE [**PRECISION**]

Remarks

The DOUBLE data type holds a double-precision floating point number. An approximate numeric data type, it is subject to rounding errors after arithmetic operations. The approximate nature of DOUBLE values means that queries using equalities should generally be avoided when comparing DOUBLE values.

DOUBLE values require 8 bytes of storage.

The value range is 2.22507385850721e–308 to 1.79769313486231e+308. Values held as DOUBLE are accurate to 15 significant digits, but may be subject to round-off error beyond the fifteenth digit.

See also

- "FLOAT data type" on page 59
- "REAL data type" on page 62
- "DECIMAL data type" on page 57
- "NUMERIC data type" on page 61
- "Numeric functions" on page 98
- "Aggregate functions" on page 93
- "Converting between numeric sets" on page 86

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

FLOAT data type

The FLOAT data type is used to store a floating point number, which can be single or double precision.

Syntax

FLOAT [**(** *precision* **)**]

Parameters

precision An integer expression that specifies the number of bits in the mantissa. A mantissa is the decimal part of a logarithm. For example, in the logarithm 5.63428, the mantissa is 0.63428. The IEEE standard 754 floating-point precision is as follows:

Supplied precision value	Decimal precision	Equivalent SQL data type	Storage size
1-24	7 decimal digits	REAL	4 bytes
25-53	15 decimal digits	DOUBLE	8 bytes

Remarks

When a column is created using the FLOAT (*precision*) data type, columns on all platforms are guaranteed to hold the values to at least the specified minimum precision. In contrast, REAL and DOUBLE do not guarantee a platform-independent minimum precision.

If *precision* is not supplied, the FLOAT data type is a single precision floating point number, equivalent to the REAL data type, and requires 4 bytes of storage.

If *precision* is supplied, the FLOAT data type is either single or double precision, depending on the value of precision specified. The cutoff between REAL and DOUBLE is platform-dependent. Single precision FLOAT values require 4 bytes of storage, and double precision FLOAT values require 8 bytes.

The FLOAT data type is an approximate numeric data type. It is subject to round-off errors after arithmetic operations. The approximate nature of FLOAT values means that queries using equalities should generally be avoided when comparing FLOAT values.

You can tune the behavior of the FLOAT data type for compatibility with Adaptive Server Enterprise, using the "float_as_double option [compatibility]" [*SQL Anywhere Server - Database Administration*].

See also

- "DOUBLE data type" on page 58
- "REAL data type" on page 62
- "DECIMAL data type" on page 57
- "NUMERIC data type" on page 61
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

INTEGER data type

The INTEGER data type is used to store integers that require 4 bytes of storage.

Syntax

[**UNSIGNED**] **INTEGER**

Remarks

The INTEGER data type is an exact numeric data type; its accuracy is preserved after arithmetic operations.

If you specify UNSIGNED, the integer can never be assigned a negative number. By default, the data type is signed.

The range for signed integers is -2^{31} to $2^{31} - 1$, or –2147483648 to 2147483647.

The range for unsigned integers is 0 to $2^{32} - 1$, or 0 to 4294967295.

INTEGER can also be specified as INT. Regardless of which syntax is used, the data type is described as INTEGER.

See also

- "BIGINT data type" on page 56
- "BIT data type" on page 57
- "SMALLINT data type" on page 62
- "TINYINT data type" on page 63
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. The UNSIGNED keyword is a vendor extension.

NUMERIC data type

The NUMERIC data types is used to store decimal numbers with *precision* total digits and with *scale* digits after the decimal point.

Syntax

NUMERIC [**(** *precision* [, *scale*] **)**]

Parameters

precision An integer expression between 1 and 127, inclusive, that specifies the number of digits in the expression. The default setting is 30.

scale An integer expression between 0 and 127, inclusive, that specifies the number of digits after the decimal point. The scale value should always be less than or equal to the precision value. The default setting is 6.

The defaults can be changed by setting database options. For information, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

Remarks

The NUMERIC data type is an exact numeric data type; its accuracy is preserved to the least significant digit after arithmetic operations.

The number of bytes required to store a decimal number can be estimated as

```
2 + int( (before+1)/2 ) + int( (after+1)/2 )
```

The function **int** takes the integer portion of its argument, and **before** and **after** are the number of significant digits before and after the decimal point. The storage is based on the value being stored, not on the maximum precision and scale allowed in the column.

NUMERIC is semantically equivalent to DECIMAL.

See also

- "FLOAT data type" on page 59
- "REAL data type" on page 62
- "DOUBLE data type" on page 58
- "DECIMAL data type" on page 57
- "Numeric functions" on page 98
- "Aggregate functions" on page 93
- "Converting between numeric sets" on page 86

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003, if the scale option is set to zero.

REAL data type

The REAL data type is used to store single-precision floating-point numbers stored in 4 bytes.

Syntax

REAL

Remarks

The REAL data type is an approximate numeric data type; it is subject to roundoff errors after arithmetic operations.

The range of values is -3.402823e+38 to 3.402823e+38, with numbers close to zero as small as 1.175495e-38. Values held as REAL are accurate to 10 significant digits, but may be subject to round-off error beyond the sixth digit.

The approximate nature of REAL values means that queries using equalities should generally be avoided when comparing REAL values

See also

- "DOUBLE data type" on page 58
- "FLOAT data type" on page 59
- "DECIMAL data type" on page 57
- "NUMERIC data type" on page 61
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

SMALLINT data type

The SMALLINT data type is used to store integers that require 2 bytes of storage.

Syntax

[**UNSIGNED**] **SMALLINT**

Remarks

The SMALLINT data type is an exact numeric data type; its accuracy is preserved after arithmetic operations. It requires 2 bytes of storage.

The range for signed SMALLINT values is -2^{15} to $2^{15} - 1$, or –32768 to 32767.

The range for unsigned SMALLINT values is 0 to $2^{16} - 1$, or 0 to 65535.

See also

- "BIGINT data type" on page 56
- "BIT data type" on page 57
- "INTEGER data type" on page 60

- "TINYINT data type" on page 63
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. The UNSIGNED keyword is a vendor extension.

TINYINT data type

The TINYINT data type is used to store unsigned integers requiring 1 byte of storage.

Syntax

[**UNSIGNED**] **TINYINT**

Remarks

The TINYINT data type is an exact numeric data type; its accuracy is preserved after arithmetic operations.

You can explicitly specify TINYINT as UNSIGNED, but the UNSIGNED modifier has no effect as the type is always unsigned.

The range for TINYINT values is 0 to $2^8 - 1$, or 0 to 255.

In embedded SQL, TINYINT columns should not be fetched into variables defined as char or unsigned char, since the result is an attempt to convert the value of the column to a string and then assign the first byte to the variable in the program. Instead, TINYINT columns should be fetched into 2-byte or 4-byte integer columns. Also, to send a TINYINT value to a database from an application written in C, the type of the C variable should be integer.

See also

- "BIGINT data type" on page 56
- "BIT data type" on page 57
- "INTEGER data type" on page 60
- "SMALLINT data type" on page 62
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

Money data types

Money data types are used for storing monetary data.

MONEY data type

The MONEY data type stores monetary data.

Syntax

MONEY

Remarks

MONEY is implemented as a domain, as NUMERIC(19,4).

See also

- "SMALLMONEY data type" on page 64
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

SMALLMONEY data type

The SMALLMONEY data type is used to store monetary data that is less than one million currency units.

Syntax

SMALLMONEY

Remarks

SMALLMONEY is implemented as a domain, as NUMERIC(10,4).

See also

- "MONEY data type" on page 64
- "Numeric functions" on page 98
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

Bit array data types

Bit arrays are used for storing bit data (0s and 1s).

A **bit array** data type is used to store an array of bits. The bit array data types supported by SQL Anywhere include VARBIT and LONG VARBIT.

LONG VARBIT data type

The LONG VARBIT data type is used to store arbitrary length bit arrays.

Syntax

LONG VARBIT

Remarks

Used to store arbitrary length array of bits (1s and 0s), or bit arrays longer than 32767 bits.

LONG VARBIT can also be specified as LONG BIT VARYING. Regardless of which syntax is used, the data type is described as LONG VARBIT.

See also

- "VARBIT data type" on page 65
- "Converting bit arrays" on page 85
- "Bit array functions" on page 94
- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

VARBIT data type

The VARBIT data type is used for storing bit arrays that are under 32767 bits in length.

Syntax

VARBIT [**(***max-length* **)**]

Parameters

max-length The maximum length of the bit array, in bits. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

VARBIT can also be specified as BIT VARYING. Regardless of which syntax is used, the data type is described as VARBIT.

See also

- “LONG VARBIT data type” on page 65
- “Converting bit arrays” on page 85
- “Bit array functions” on page 94
- “Aggregate functions” on page 93

Standards and compatibility

- **SQL/2003** Vendor extension.

Date and time data types

This section examines how date values are handled internally by SQL Anywhere, and how SQL Anywhere handles ambiguous date information, such as the conversion of a two digit year string value.

The following list provides a quick overview of how dates are handled:

- SQL Anywhere always returns correct values for any legal arithmetic and logical operations on dates, regardless of whether the calculated values span different centuries.
- At all times, the SQL Anywhere internal storage of dates explicitly includes the century portion of a year value.
- The operation of SQL Anywhere is unaffected by any return value, including the current date.
- Date values can always be output in full century format.

How dates are stored

Dates containing year values are used internally and stored in SQL Anywhere databases using either of the following data types:

Data type	Contains	Stored in	Range of possible values
DATE	Calendar date (year, month, day)	4-bytes	0001-01-01 to 9999-12-31
TIMESTAMP	Time stamp (year, month, day, hour minute, second, and fraction of second accurate to 6 decimal places)	8-bytes	0001-01-01 to 9999-12-31 (precision of the time portion of TIMESTAMP is dropped prior to 1600-02-28 23:59:59 and after 7911-01-01 00:00:00)

For more information on SQL Anywhere date and time data types see "Date and time data types" on page 67.

Sending dates and times to the database

Date and times may be sent to the database in one of the following ways:

- Using any interface, as a string
- Using ODBC, as a TIMESTAMP structure
- Using embedded SQL, as a SQLDATETIME structure

When a time is sent to the database as a string (for the TIME data type) or as part of a string (for TIMESTAMP or DATE data types), the hours, minutes, and seconds must be separated by colons in the format *hh*:*mm*:*ss.sss*, but can appear anywhere in the string. The following are valid and unambiguous strings for specifying times:

```
21:35 -- 24 hour clock if no am or pm specified
10:00pm -- pm specified, so interpreted as 12 hour clock
10:00 -- 10:00am in the absence of pm
10:23:32.234 -- seconds and fractions of a second included
```

When a date is sent to the database as a string conversion to a DATE or TIMESTAMP data type is automatic. The string can be supplied in one of two ways:

- As a string of format *yyyy/mm/dd* or *yyyy-mm-dd*, which is interpreted unambiguously by the database.
- As a string interpreted according to the date_order database option. See "date_order option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Transact-SQL string-to-date/time conversions

Converting strings to date and time data types.

If a string containing only a time value (no date) is converted to a date/time data type, SQL Anywhere uses the current date.

If the fraction portion of a time is less than 3 digits, SQL Anywhere interprets the value the same way regardless of the whether it is preceded by a period or a colon: one digit means tenths, two digits mean hundredths, and three digits mean thousandths.

Examples

SQL Anywhere converts the milliseconds value in the same manner regardless of the separator.

```
12:34:56.7 to 12:34:56.700
12:34:56:7 to 12:34:56.700
12.34.56.78 to 12:34:56.780
12.34.56:78 to 12:34:56.780
12:34:56.789 to 12:34:56.789
12:34:56:789 to 12:34:56.789
```

Retrieving dates and times from the database

Dates and times may be retrieved from the database in one of the following ways:

- Using any interface, as a string
- Using ODBC, as a TIMESTAMP structure
- Using embedded SQL, as a SQLDATETIME structure

When a date or time is retrieved as a string, it is retrieved in the format specified by the database options date_format, time_format and timestamp_format. For descriptions of these options, see "SET OPTION statement" on page 664.

For information on functions that deal with dates and times, see "Date and time functions" on page 94. The following arithmetic operators are allowed on dates:

- **timestamp + integer** Add the specified number of days to a date or timestamp.
- **timestamp - integer** Subtract the specified number of days from a date or timestamp.
- **date - date** Compute the number of days between two dates or timestamps.
- **date + time** Create a timestamp combining the given date and time.

Leap Years

SQL Anywhere uses a globally accepted algorithm for determining which years are leap years. Using this algorithm, a year is considered a leap year if it is divisible by four, unless the year is a century date (such as the year 1900), in which case it is a leap year only if it is divisible by 400.

SQL Anywhere handles all leap years correctly. For example, the following SQL statement results in a return value of "Tuesday":

```
SELECT DAYNAME('2000-02-29')
```

SQL Anywhere accepts February 29, 2000—a leap year—as a date, and using this date determines the day of the week.

However, the following statement is rejected by SQL Anywhere:

```
SELECT DAYNAME('2001-02-29')
```

This statement results in an error (cannot convert '2001-02-29' to a date) because February 29th does not exist in the year 2001.

Comparing dates and times

By default, values stored as DATE do not have any hour or minute values, and so comparison of dates is straightforward.

The DATE data type can also contain a time, which introduces complications when comparing dates. If the time is not specified when a date is entered into the database, the time defaults to 0:00 or 12:00am (midnight). Any date comparisons with this option setting compare the times as well as the date itself. A database date value of 1999-05-23 10:00 is not equal to the constant 1999-05-23. The DATEFORMAT function or one of the other date functions can be used to compare parts of a date and time field. For example,

```
DATEFORMAT(invoice_date,'yyyy/mm/dd') = '1999/05/23'
```

If a database column requires only a date, client applications should ensure that times are not specified when data is entered into the database. This way, comparisons with date-only strings will work as expected.

If you want to compare a date to a string *as a string*, you must use the DATEFORMAT function or CAST function to convert the date to a string before comparing.

Using unambiguous dates and times

Dates in the format *yyyy/mm/dd* or *yyyy-mm-dd* are always recognized unambiguously as dates, regardless of the date_order setting. Other characters can be used as separators instead of "**/**" or "**-**"; for example, "**?**", a space character, or "**,**". You should use this format in any context where different users may be employing different date_order settings. For example, in stored procedures, use of the unambiguous date format prevents misinterpretation of dates according to the user's date_order setting.

Also, a string of the form *hh:mm:ss:ssss* is interpreted unambiguously as a time.

For combinations of dates and times, any unambiguous date and any unambiguous time yield an unambiguous date-time value. Also, the form *yyyy-mm-dd hh.mm.ss.sss*

is an unambiguous date-time value. Periods can be used in the time only in combination with a date.

In other contexts, a more flexible date format can be used. SQL Anywhere can interpret a wide range of strings as dates. The interpretation depends on the setting of the database option date_order. The date_order database option can have the value *MDY*, *YMD*, or *DMY* (see "SET OPTION statement" on page 664). For example, the following statement sets the date_order option to *DMY*:

```
SET OPTION date_order = 'DMY' ;
```

The default date_order setting is YMD. The ODBC driver sets the date_order option to YMD whenever a connection is made. The value can still be changed using the SET TEMPORARY OPTION statement.

The database option date_order determines whether the string 10/11/12 is interpreted by the database as November 12, 2010; October 11, 2012; or November 10, 2012. The year, month, and day of a date string should be separated by some character (/, -, or space) and appear in the order specified by the date_order option.

The year can be supplied as either 2 or 4 digits. The value of the nearest_century option affects the interpretation of 2-digit years: 2000 is added to values less than nearest_century and 1900 is added to all other values. The default value of this option is 50. Thus, by default, 50 is interpreted as 1950 and 49 is interpreted 2049.

The month can be the name or number of the month. The hours and minutes are separated by a colon, but can appear anywhere in the string.

Notes

- It is recommended that you always specify the year using the four-digit format.
- With an appropriate setting of date_order, the following strings are all valid dates:

```
99-05-23 21:35
99/5/23
1999/05/23
May 23 1999
```



```
23-May-1999
Tuesday May 23, 1999 10:00pm
```

- If a string contains only a partial date specification, default values are used to fill out the date. The following defaults are used:
  - **year** This year
  - **month** No default
  - **day** 1 (useful for month fields; for example, May 1999 will be the date 1999-05-01 00:00)
  - **hour, minute, second, fraction** 0

DATE data type

The DATE data type is used to store calendar dates, such as a year, month and day.

Syntax

DATE

Remarks

The year can be from the year 0001 to 9999. The minimum date in SQL Anywhere is 0001-01-01 00:00:00.

For historical reasons, a DATE column can also contain an hour and minute. The TIMESTAMP data type is recommended for anything with hours and minutes.

The format in which DATE values are retrieved by applications is controlled by the date_format setting. For example, a date value representing the 19th of July, 2003 may be returned to an application as `2003/07/19`, as `Jul 19, 2003`, or as one of a number of other possibilities.

The way in which a string is interpreted by the database server as a date is controlled by the date_order option. For example, depending on the date_order setting, a value of `02/05/2002` supplied by an application for a DATE value may be interpreted in the database as the 2nd of May or the 5th of February.

A DATE value requires 4 bytes of storage.

See also

- "date_format option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "date_order option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "DATETIME data type" on page 72
- "SMALLDATETIME data type" on page 72
- "TIMESTAMP data type" on page 73
- "Date and time functions" on page 94

Standards and compatibility

- **SQL/2003** Vendor extension.

DATETIME data type

The DATETIME data type is a domain, implemented as TIMESTAMP, used to store date and time information.

Syntax

DATETIME

See also

- "DATE data type" on page 71
- "SMALLDATETIME data type" on page 72
- "TIMESTAMP data type" on page 73
- "Date and time functions" on page 94

Standards and compatibility

- **SQL/2003** Vendor extension.

SMALLDATETIME data type

The SMALLDATETIME data type is a domain, implemented as TIMESTAMP, used to store date and time information.

Syntax

SMALLDATETIME

See also

- "DATE data type" on page 71
- "DATETIME data type" on page 72
- "TIMESTAMP data type" on page 73
- "Date and time functions" on page 94

Standards and compatibility

- **SQL/2003** Vendor extension.

TIME data type

The TIME data type is used to store the time of day, containing hour, minute, second and fraction of a second.

Syntax

TIME

Remarks

The fraction is stored to 6 decimal places. A TIME value requires 8 bytes of storage. (ODBC standards restrict TIME data type to an accuracy of seconds. For this reason you should not use TIME data types in WHERE clause comparisons that rely on a higher accuracy than seconds.)

See also

- "TIMESTAMP data type" on page 73
- "Date and time functions" on page 94

Standards and compatibility

- **SQL/2003** Vendor extension.

TIMESTAMP data type

The TIMESTAMP data type is used to store a point in time containing year, month, day, hour, minute, second and fraction of a second.

Syntax

TIMESTAMP

Remarks

The fraction is stored to 6 decimal places. A TIMESTAMP value requires 8 bytes of storage.

Although the range of possible dates for the TIMESTAMP data type is the same as the DATE type (covering years 0001 to 9999), the useful range of TIMESTAMP date types is from 1600-02-28 23:59:59 to 7911-01-01 00:00:00. Prior to, and after this range the time portion of the TIMESTAMP may be incomplete.

See also

- "TIME data type" on page 72
- "Date and time functions" on page 94

Standards and compatibility

- **SQL/2003** Vendor extension.

Binary data types

Binary data types are used for storing binary data, including images and other types of information that are not interpreted by the database.

BINARY data type

The BINARY data type is used to store binary data of a specified maximum length (in bytes).

Syntax

BINARY [**(** *max-length* **)**]

Parameters

max-length The maximum length of the value, in bytes. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

During comparisons, BINARY values are compared exactly byte for byte. This differs from the CHAR data type, where values are compared using the collation sequence of the database. If one binary string is a prefix of the other, the shorter string is considered to be less than the longer string.

Unlike CHAR values, BINARY values are not transformed during character set conversion.

BINARY is semantically equivalent to VARBINARY. It is a variable-length type. In other database management systems, BINARY is a fixed-length type.

See also

- "VARBINARY data type" on page 76
- "LONG BINARY data type" on page 75
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

IMAGE data type

The IMAGE data type is used to store binary data of arbitrary length.

Syntax

IMAGE

Remarks

IMAGE is implemented as a domain, as LONG BINARY.

See also

- "LONG BINARY data type" on page 75
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

LONG BINARY data type

The LONG BINARY data type is used to store binary data of arbitrary length.

Syntax

LONG BINARY

Remarks

The maximum size is 2 GB.

See also

- "BINARY data type" on page 74
- "VARBINARY data type" on page 76
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

UNIQUEIDENTIFIER data type

The UNIQUEIDENTIFIER data type is used to store UUID (also known as GUID) values.

Syntax

UNIQUEIDENTIFIER

Remarks

The UNIQUEIDENTIFIER data type is typically used for a primary key or other unique column to hold UUID (Universally Unique Identifier) values that uniquely identify rows. The NEWID function generates UUID values in such a way that a value produced on one computer will not match a UUID produced on another computer. UNIQUEIDENTIFIER values generated using NEWID can therefore be used as keys in a synchronization environment.

For example:

```
CREATE TABLE T1 (
    pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    c1 INT )
```


UUID values are also referred to as GUIDs (Globally Unique Identifier). By default, UUID values contain hyphens so they are compatible with other RDBMSs. You can change this by setting the uuid_has_hyphens option to Off.

SQL Anywhere automatically converts UNIQUEIDENTIFIER values between string and binary values as needed.

UNIQUEIDENTIFIER values are stored as BINARY(16), but are described to client applications as BINARY(36). This description ensures that if the client fetches the value as a string, it has allocated sufficient space for the result. For ODBC client applications, uniqueidentifier values appear as a SQL_GUID type.

See also

- "The NEWID default" [*SQL Anywhere Server - SQL Usage*]
- "NEWID function [Miscellaneous]" on page 204
- "UUIDTOSTR function [String]" on page 274
- "STRTOUUID function [String]" on page 261
- "uuid_has_hyphens option [database]" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

VARBINARY data type

The VARBINARY data type is used to store binary data of a specified maximum length (in bytes).

Syntax

VARBINARY [**(** *max-length* **)**]

Parameters

max-length The maximum length of the value, in bytes. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

During comparisons, VARBINARY values are compared exactly byte for byte. This differs from the CHAR data type, where values are compared using the collation sequence of the database. If one binary string is a prefix of the other, the shorter string is considered to be less than the longer string.

Unlike CHAR values, VARBINARY values are not transformed during character set conversion.

VARBINARY can also be specified as BINARY VARYING. Regardless of which syntax is used, the data type is described as VARBINARY.

See also

- "BINARY data type" on page 74
- "LONG BINARY data type" on page 75
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Domains

Domains are aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere, but you can add more of your own.

Domains, also called **user-defined data types**, allow columns throughout a database to be automatically defined on the same data type, with the same NULL or NOT NULL condition, with the same DEFAULT setting, and with the same CHECK condition. Domains encourage consistency throughout the database and can eliminate some types of errors.

Simple domains

Domains are created using the CREATE DOMAIN statement. For a full description of the syntax, see "CREATE DOMAIN statement" on page 386.

The following statement creates a data type named street_address, which is a 35-character string.

```
CREATE DOMAIN street_address CHAR( 35 )
```

CREATE DATATYPE can be used as an alternative to CREATE DOMAIN, but is not recommended.

Resource authority is required to create data types. Once a data type is created, the user ID that executed the CREATE DOMAIN statement is the owner of that data type. Any user can use the data type. Unlike with other database objects, the owner name is never used to prefix the data type name.

The **street_address** data type may be used in exactly the same way as any other data type when defining columns. For example, the following table with two columns has the second column as a **street_address** column:

```
CREATE TABLE twocol (
    id INT,
    street street_address
)
```

Domains can be dropped by their owner or by a user with DBA authority, using the DROP DOMAIN statement:

```
DROP DOMAIN street_address
```

This statement can be carried out only if the data type is not used in any table in the database. If you attempt to drop a domain that is in use, the message "Primary key for row in table 'SYSUSERTYPE' is referenced in another table" appears.

Constraints and defaults with domains

Many of the attributes associated with columns, such as allowing NULL values, having a DEFAULT value, and so on, can be built into a domain. Any column that is defined on the data type automatically inherits the NULL setting, CHECK condition, and DEFAULT values. This allows uniformity to be built into columns with a similar meaning throughout a database.

For example, many primary key columns in the SQL Anywhere sample database are integer columns holding ID numbers. The following statement creates a data type that may be useful for such columns:


```
CREATE DOMAIN id INT
NOT NULL
DEFAULT AUTOINCREMENT
CHECK( @col > 0 );
```

Any column created using the data type **id** is not allowed to hold NULLs, defaults to an auto-incremented value, and must hold a positive number. Any identifier could be used instead of *col* in the *@col* variable.

The attributes of the data type can be overridden if needed by explicitly providing attributes for the column. A column created on data type **id** with NULL values explicitly allowed does allow NULLs, regardless of the setting in the **id** data type.

Compatibility

- **Named constraints and defaults** In SQL Anywhere, domains are created with a base data type, and optionally a NULL or NOT NULL condition, a default value, and a CHECK condition. Named constraints and named defaults are not supported.
- **Creating data types** In SQL Anywhere, you can use the sp_addtype system procedure to add a domain, or you can use the CREATE DOMAIN statement.

Data type conversions

Type conversions can happen automatically, or they can be explicitly requested using the CAST or CONVERT function. The following functions can also be used to force type conversions :

- **DATE function** Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.
- **STRING function** This function is equivalent to `CAST( value AS LONG VARCHAR)`.
- **VALUE+0.0** Equivalent to CAST(value AS DECIMAL).

The following list is a high-level view of autmatic data type conversions:

- If a string is used in a numeric expression or as an argument to a function that expects a numeric argument, the string is converted to a number.
- If a number is used in a string expression or as a string function argument, it is converted to a string before being used.
- All date constants are specified as strings. The string is automatically converted to a date before use.

There are certain cases where the automatic database conversions are not appropriate. For example, the automatic data type conversion fails in the example below.

```
'12/31/90' + 5
'a' > 0
```

See also

- “Data type conversion functions” on page 94
- “DATE function [Date and time]” on page 136
- “STRING function [String]” on page 260
- “CAST function [Data type conversion]” on page 115

Comparisons between data types

When a comparison (such as =) is performed between arguments with different data types, one or more arguments must be converted so that the comparison operation is done using one data type.

Some rules may lead to conversions that fail, or lead to unexpected results from the comparison. In these cases, you should explicitly convert one of the arguments using CAST or CONVERT.

You can override these conversion rules by explicitly casting arguments to another type. For example, if you want to compare a DATE and a CHAR as a CHAR, then you need to explicitly cast the DATE to a CHAR. See “CAST function [Data type conversion]” on page 115.

Substitution characters

When a character cannot be represented in the character set into which it is being converted, a substitution character set is used instead. Conversions of this type are considered **lossy**; that is, the original character is lost if it cannot be represented in the destination character set.

Also, not only may different character sets may have different substitution characters, but the substitution character for one character set may be a non-substitution character in another character set. This is important to understand when multiple conversions are performed on a character because the final character may not appear as the expected substitution character of the destination character set.

For example, suppose that the client character set is Windows-1252, and the database character set is ISO_8859-1:1987, the U.S. default for some versions of Unix. Then, suppose a non-Unicode client application (for example, embedded SQL) attempts to insert the euro symbol into a CHAR, VARCHAR, or LONG VARCHAR column. Since the character does not exist in the CHAR character set, the substitution character for ISO_8859-1:1987, 0x1A, is inserted.

Now, if this same ISO_8859-1:1987 substitution character is then fetched to a UTF-16 value (for example, by doing a `SELECT * FROM t` into a SQL_C_WCHAR bound column in ODBC), this character becomes the UTF-16 character 0x001A. However, this is not the substitution character, defined for UTF-16, for a euro. This example illustrates that even if your data contains substitution characters, those characters, due to multiple conversions, may be inconsistent with the characters defined for the target character set.

Therefore, it is important to understand and test how substitution characters may be used when converting between multiple character sets.

The on_charset_conversion_failure option determines the behavior during conversion when a character cannot be represented in the destination character set. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "Data type conversions" on page 80
- "Comparisons between CHAR and NCHAR" on page 81
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Comparisons between CHAR and NCHAR

When a comparison is performed between a CHAR type (CHAR, VARCHAR, LONG VARCHAR) and an NCHAR type (NCHAR, NVARCHAR, LONG NVARCHAR), SQL Anywhere uses inference rules to determine whether the NCHAR value can, and should, be **coerced** to the CHAR type. A value can be coerced if it is a literal constant, a variable, a host variable, or a complex expression not based on a column reference. Generally, when an NCHAR value is compared to a CHAR column, the comparison is performed as CHAR if the NCHAR value can be coerced to CHAR; otherwise, it is performed as NCHAR.

Following are the inference rules, in the order in which they are applied:

- If there is any non-coercible NCHAR value, then all CHAR values are converted to NCHAR, and the comparison is done as NCHAR.

- Else, if there is any non-coercible CHAR value, then all NCHAR values are converted to CHAR, and the comparison is done as CHAR.

  It is important to consider the setting for the on_charset_conversion_failure option if you anticipate NCHAR to CHAR conversions since this option controls behavior if an NCHAR character cannot be represented in the CHAR character set. For further explanation, see "Converting NCHAR to CHAR" on page 84.

- Else, if there is a mix of coercible CHAR and NCHAR values (that is, all values are coercible), then all CHAR values are converted to NCHAR, and the comparison is done as NCHAR.

Examples

The condition `Employees.GivenName = N'Susan'` compares a CHAR column (Employees.GivenName) to the literal N'Susan'. The value N'Susan' is coerced to CHAR, and the comparison is performed as if it had been written as:

```
Employees.GivenName  = CAST( N'Susan' AS CHAR )
```

Alternatively, the condition `Employees.GivenName = T.nchar_column` would find that the value T.nchar_column can not be coerced to CHAR. The comparison would be performed as if it were written as follows, and an index on Employees.GivenName can not be used:

```
CAST( Employees.GivenName AS NCHAR ) = T.nchar_column
```

See also

- "Converting NCHAR to CHAR" on page 84
- "Substitution characters" on page 81
- "CAST function [Data type conversion]" on page 115
- "CONVERT function [Data type conversion]" on page 125
- "CAST function [Data type conversion]" on page 115
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Comparisons between numeric data types

SQL Anywhere uses the following rules when comparing numeric data types. The rules are examined in the order listed, and the first rule that applies is used:

1. If one argument is TINYINT and the other is INTEGER, convert both to INTEGER and compare.
2. If one argument is TINYINT and the other is SMALLINT, convert both to SMALLINT and compare.
3. If one argument is UNSIGNED SMALLINT and the other is INTEGER, convert both to INTEGER and compare.
4. If the data types of the arguments have a common super type, convert to the common super type and compare. The super types are the final data type in each of the following lists:

   - BIT ► TINYINT ► UNSIGNED SMALLINT ► UNSIGNED INTEGER ► UNSIGNED BIGINT ► NUMERIC

- SMALLINT ► INTEGER ► BIGINT ► NUMERIC
- REAL ► DOUBLE
- CHAR ► LONG VARCHAR
- BINARY ► LONG BINARY

For example, if the two arguments are of types BIT and TINYINT, they are converted to NUMERIC.

Comparisons between time and date data types

SQL Anywhere uses the following rules when comparing time and date data types. The rules are examined in the order listed, and the first rule that applies is used:

1. If the data type of either argument is TIME, convert both to TIME and compare.
2. If either data type has the type DATE or TIMESTAMP, convert to both to TIMESTAMP and compare.

   For example, if the two arguments are of type REAL and DATE, they are both converted to TIMESTAMP.
3. If one argument has NUMERIC data type and the other has FLOAT, convert both to DOUBLE and compare.

Other comparisons

1. If the data types are a mixture of CHAR (such as CHAR, VARCHAR, LONG VARCHAR, and so on, but not NCHAR types), convert to LONG VARCHAR and compare.
2. If the data type of any argument is UNIQUEIDENTIFIER, convert to UNIQUEIDENTIFIER and compare.
3. If the data type of any argument is a bit array (VARBIT or LONG VARBIT), convert to LONG VARBIT and compare.
4. If one argument has CHARACTER data type and the other has BINARY data type, convert to BINARY and compare.
5. If one argument is a CHAR type, and the other argument is an NCHAR type, use predefined inference rules. See “Comparisons between CHAR and NCHAR” on page 81.
6. If no rule exists, convert to NUMERIC and compare.

   For example, if the two arguments have REAL and CHAR data types, they are both converted to NUMERIC.

Converting NCHAR to CHAR

NCHAR to CHAR conversions can occur as part of a comparison of CHAR and NCHAR data, or when specifically requested. This type of conversion is considered **lossy** because there there are some NCHAR characters that can not be represented in the CHAR type. When these characters are present in the NCHAR data, a substitution character from the CHAR character set is used instead. For single-byte character sets, this is usually hex 1A.

Depending on the setting of the on_charset_conversion_failure option, when a character cannot be converted, one of the following can happen:

- a substitute character is used, and no warning is issued
- a substitute character is used, and a warning is issued
- an error is returned

Therefore, it is important to consider this option when converting from NCHAR to CHAR. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "Comparisons between CHAR and NCHAR" on page 81
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Converting NULL constants to NUMERIC and string types

When converting a NULL constant to a NUMERIC, or to a string type such as CHAR, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, and LONG BINARY the size is set to 0. For example:

`SELECT CAST( NULL AS CHAR )` returns CHAR(0)

`SELECT CAST( NULL AS NUMERIC )` returns NUMERIC(1,0)

Converting dates to strings

SQL Anywhere provides several functions for converting SQL Anywhere date and time values into a wide variety of strings and other expressions. It is possible in converting a date value into a string to reduce the year portion into a two-digit number representing the year, thereby losing the century portion of the date.

Wrong century values

Consider the following statement, which incorrectly converts a string representing the date January 1, 2000 into a string representing the date January 1, 1900 even though no database error occurs.

```
SELECT DATEFORMAT (
            DATEFORMAT('2000-01-01', 'Mmm dd/yy' ),
            'yyyy-Mmm-dd' )
    AS Wrong_year;
```


SQL Anywhere automatically and correctly converts the unambiguous date string 2000-01-01 into a date value. However, the 'Mmm dd/yy' formatting of the inner, or nested, DATEFORMAT function drops the century portion of the date when it is converted back to a string and passed to the outer DATEFORMAT function.

Because the database option nearest_century in this case is set to 0, the outer DATEFORMAT function converts the string representing a date with a two-digit year value into a year between 1900 and 1999.

For more information on date and time functions, see "Date and time functions" on page 94.

Converting bit arrays

Converting integers to bit arrays

When converting an integer to a bit array, the length of the bit array is the number of bits in the integer type, and the bit array's value is the integer's binary representation. The most significant bit of the integer becomes the first bit of the array.

Examples

- `SELECT CAST( CAST( 1 AS BIT ) AS VARBIT )` returns a VARBIT(1) containing 1.
- `SELECT CAST( CAST( 8 AS TINYINT ) AS VARBIT )` returns a VARBIT(8) containing 00001000.
- `SELECT CAST( CAST( 194 AS INTEGER ) AS VARBIT )` returns a VARBIT(32) containing 00000000000000000000000011000010.

Converting binary to bit arrays

When converting a binary type of length *n* to a bit array, the length of the array is *n* * 8 bits. The first 8 bits of the bit array become the first byte of the binary value. The most significant bit of the binary value becomes the first bit in the array. The next 8 bits of the bit array become the second byte of the binary value, and so on.

Examples

- `SELECT CAST( 0x8181 AS VARBIT )` returns a VARBIT(16) containing 1000000110000001.

Converting characters to bit arrays

When converting a character data type of length *n* to a bit array, the length of the array is *n* bits. Each character must be either '0' or '1' and the corresponding bit of the array is assigned the value 0 or 1.

Examples

- `SELECT CAST( '001100' AS VARBIT )` returns a VARBIT(6) containing 001100.

Converting bit arrays to integers

When converting a bit array to an integer data type, the bit array's binary value is interpreted according to the storage format of the integer type, using the most significant bit first.

Examples

- `SELECT CAST( CAST( '11000010' AS VARBIT ) AS INTEGER )` returns 194 (11000010_2 = 0xC2 = 194).

Converting bit arrays to binary

When converting a bit array to a binary, the first 8 bits of the array become the first byte of the binary value. The first bit of the array becomes the most significant bit of the binary value. The next 8 bits are used as the second byte, and so on. If the length of the bit array is not a multiple of 8, then extra zeroes are used to fill the least significant bits of the last byte of the binary value.

Examples

- `SELECT CAST( CAST( '1111' AS VARBIT ) AS BINARY )` returns 0xF0 (1111_2 becomes 11110000_2 = 0xF0).
- `SELECT CAST( CAST( '0011000000110001' AS VARBIT ) AS BINARY )` returns 0x3031 (0011000000110001_2 = 0x3031).

Converting bit arrays to characters

When converting a bit array of length *n* bits to a character data type, the length of the result is *n* characters. Each character in the result is either '0' or '1', corresponding to the bit in the array.

Examples

- `SELECT CAST( CAST( '01110' AS VARBIT ) AS VARCHAR )` returns the character string '01110'.

Converting between numeric sets

When converting a DOUBLE type to a NUMERIC type, precision is maintained for the first 15 significant digits.

See also

- “CAST function [Data type conversion]” on page 115
- “CONVERT function [Data type conversion]” on page 125
- “CAST function [Data type conversion]” on page 115

Ambiguous string to date conversions

SQL Anywhere automatically converts a string into a date when a date value is expected, even if the year is represented in the string by only two digits.

If the century portion of a year value is omitted, the method of conversion is determined by the nearest_century database option.

The nearest_century database option is a numeric value that acts as a break point between 19YY date values and 20YY date values.

Two-digit years less than the nearest_century value are converted to 20yy, while years greater than or equal to the value are converted to 19yy.

If this option is not set, the default setting of 50 is assumed. Thus, two-digit year strings are understood to refer to years between 1950 and 2049.

This nearest_century option was introduced in SQL Anywhere Version 5.5. In version 5.5, the default setting was 0.

Ambiguous date conversion example

The following statement creates a table that can be used to illustrate the conversion of ambiguous date information in SQL Anywhere.

```
CREATE TABLE T1 (C1 DATE);
```

The table T1 contains one column, C1, of the type DATE.

The following statement inserts a date value into the column C1. SQL Anywhere automatically converts a string that contains an ambiguous year value, one with two digits representing the year but nothing to indicate the century.

```
INSERT INTO T1 VALUES('00-01-01');
```

By default, the nearest_century option is set to 50, thus SQL Anywhere converts the above string into the date 2000-01-01. The following statement verifies the result of this insert.

```
SELECT * FROM T1;
```

Changing the nearest_century option using the following statement alters the conversion process.

```
SET OPTION nearest_century = 0;
```

When nearest_century option is set to 0, executing the previous insert using the same statement will create a different date value:

```
INSERT INTO T1 VALUES('00-01-01');
```

The above statement now results in the insertion of the date 1900-01-01. Use the following statement to verify the results.

```
SELECT * FROM T1;
```


Java and SQL data type conversion

Data type conversion between Java types and SQL types is required for both Java stored procedures and JDBC applications. Java to SQL and SQL to Java data type conversions are carried out according to the JDBC standard. The conversions are described in the following tables.

Java to SQL data type conversion

Java type	SQL type
String	CHAR
String	VARCHAR
String	TEXT
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	MONEY
java.math.BigDecimal	SMALLMONEY
boolean	BIT
byte	TINYINT
Short	SMALLINT
Int	INTEGER
long	INTEGER
float	REAL
double	DOUBLE
byte[ ]	VARBINARY
byte[ ]	IMAGE
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.lang.Double	DOUBLE
java.lang.Float	REAL
java.lang.Integer	INTEGER

Java type	SQL type
java.lang.Long	INTEGER

SQL to Java data type conversion

SQL type	Java type
CHAR	String
VARCHAR	String
TEXT	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
MONEY	java.math.BigDecimal
SMALLMONEY	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[ ]
VARBINARY	byte[ ]
LONG VARBINARY	byte[ ]
IMAGE	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

CHAPTER 3

SQL Functions

Contents

Introduction to SQL functions 92
Function types 93
Alphabetical list of functions 103

Introduction to SQL functions

Functions are used to return information from the database. They are allowed anywhere an expression is allowed.

Functions use the same syntax conventions used by SQL statements. For a complete list of syntax conventions, see “Syntax conventions” on page 297.

Function types

This section groups the available function by type.

Aggregate functions

Aggregate functions summarize data over a group of rows from the database. The groups are formed using the GROUP BY clause of the SELECT statement. Aggregate functions are allowed only in the select list and in the HAVING and ORDER BY clauses of a SELECT statement.

List of functions

The following aggregate functions are available:

- "AVG function [Aggregate]" on page 107
- "BIT_AND function [Aggregate]" on page 110
- "BIT_OR function [Aggregate]" on page 111
- "BIT_XOR function [Aggregate]" on page 112
- "COVAR_POP function [Aggregate]" on page 131
- "COVAR_SAMP function [Aggregate]" on page 132
- "COUNT function [Aggregate]" on page 129
- "CORR function [Aggregate]" on page 127
- "FIRST_VALUE function [Aggregate]" on page 165
- "GROUPING function [Aggregate]" on page 171
- "LAST_VALUE function [Aggregate]" on page 187
- "LIST function [Aggregate]" on page 192
- "MAX function [Aggregate]" on page 198
- "MIN function [Aggregate]" on page 199
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_R2 function [Aggregate]" on page 227
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "SET_BITS function [Aggregate]" on page 245
- "STDDEV function [Aggregate]" on page 257
- "STDDEV_POP function [Aggregate]" on page 257
- "STDDEV_SAMP function [Aggregate]" on page 258
- "SUM function [Aggregate]" on page 264
- "VAR_POP function [Aggregate]" on page 275
- "VAR_SAMP function [Aggregate]" on page 276
- "VARIANCE function [Aggregate]" on page 278
- "XMLAGG function [Aggregate]" on page 280

Bit array functions

Bit array functions allow you to perform tasks on bit arrays. The following bit array functions are available:

- "BIT_AND function [Aggregate]" on page 110
- "BIT_OR function [Aggregate]" on page 111
- "BIT_XOR function [Aggregate]" on page 112
- "BIT_LENGTH function [Bit array]" on page 109
- "BIT_SUBSTR function [Bit array]" on page 109
- "COUNT_SET_BITS function [Bit array]" on page 130
- "GET_BIT function [Bit array]" on page 167
- "SET_BIT function [Bit array]" on page 244
- "SET_BITS function [Aggregate]" on page 245

For information about bitwise operators, see "Bitwise operators" on page 13.

Ranking functions

Ranking functions let you compute a rank value for each row in a result set based on an ordering specified in the query.

- "CUME_DIST function [Ranking]" on page 135
- "DENSE_RANK function [Ranking]" on page 151
- "PERCENT_RANK function [Ranking]" on page 213
- "RANK function [Ranking]" on page 221

Data type conversion functions

Data type conversion functions are used to convert arguments from one data type to another, or to test whether they can be converted.

List of functions

The following data type conversion functions are available:

- "CAST function [Data type conversion]" on page 115
- "CONVERT function [Data type conversion]" on page 125
- "HEXTOINT function [Data type conversion]" on page 173
- "INTTOHEX function [Data type conversion]" on page 184
- "ISDATE function [Data type conversion]" on page 185
- "ISNUMERIC function [Miscellaneous]" on page 186

Date and time functions

Date and time functions perform operations on date and time data types or return date or time information.

In this chapter, the term **datetime** is used to mean date or time or timestamp. The specific data type DATETIME is indicated as DATETIME.

For more information on datetime data types, see "Date and time data types" on page 67.

List of functions

The following date and time functions are available:

- "DATE function [Date and time]" on page 136
- "DATEADD function [Date and time]" on page 137
- "DATEDIFF function [Date and time]" on page 137
- "DATEFORMAT function [Date and time]" on page 139
- "DATENAME function [Date and time]" on page 139
- "DATEPART function [Date and time]" on page 140
- "DATETIME function [Date and time]" on page 141
- "DAY function [Date and time]" on page 141
- "DAYNAME function [Date and time]" on page 141
- "DAYS function [Date and time]" on page 142
- "DOW function [Date and time]" on page 153
- "GETDATE function [Date and time]" on page 169
- "HOUR function [Date and time]" on page 174
- "HOURS function [Date and time]" on page 175
- "MINUTE function [Date and time]" on page 199
- "MINUTES function [Date and time]" on page 200
- "MONTH function [Date and time]" on page 202
- "MONTHNAME function [Date and time]" on page 202
- "MONTHS function [Date and time]" on page 203
- "NOW function [Date and time]" on page 210
- "QUARTER function [Date and time]" on page 218
- "SECOND function [Date and time]" on page 242
- "SECONDS function [Date and time]" on page 243
- "TODAY function [Date and time]" on page 268
- "WEEKS function [Date and time]" on page 279
- "YEAR function [Date and time]" on page 286
- "YEARS function [Date and time]" on page 286
- "YMD function [Date and time]" on page 288

Date parts

Many of the date functions use dates built from **date parts**. The following table displays allowed values of date parts.

Date part	Abbreviation	Values
Year	yy	1–9999
Quarter	qq	1–4

Date part	Abbreviation	Values
Month	mm	1–12
Week	wk	1–54. Weeks begin on Sunday.
Day	dd	1–31
Dayofyear	dy	1–366
Weekday	dw	1–7 (Sunday = 1, …, Saturday = 7)
Hour	hh	0–23
Minute	mi	0–59
Second	ss	0–59
Millisecond	ms	0–999
Calyearofweek	cyr	Integer. The year in which the week begins. The week containing the first few days of the year may have started in the previous year, depending on the weekday on which the year started. Years starting on Monday through Thursday have no days that are part of the previous year, but years starting on Friday through Sunday start their first week on the first Monday of the year.
Calweekofyear	cwk	1–54. The week number within the year that contains the specified date.
Caldayofweek	cdw	1–7. (Monday = 1, …, Sunday = 7)

User-defined functions

There are two mechanisms for creating user-defined functions in SQL Anywhere. You can use the SQL language to write the function, or you can use Java.

User-defined functions in SQL

You can implement your own functions in SQL using the "CREATE FUNCTION statement" on page 399. The RETURN statement inside the CREATE FUNCTION statement determines the data type of the function.

Once a SQL user-defined function is created, it can be used anywhere a built-in function of the same data type is used.

For more information on creating SQL functions, see "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*].

User-defined functions in Java

Java classes provide a more powerful and flexible way of implementing user-defined functions, with the additional advantage that they can be moved from the database server to a client application if desired.

Any class method of an installed Java class can be used as a user-defined function anywhere a built-in function of the same data type is used.

Instance methods are tied to particular instances of a class, and so have different behavior from standard user-defined functions.

For more information on creating Java classes, and on class methods, see "Creating a class" [*SQL Anywhere Server - Programming*].

Miscellaneous functions

Miscellaneous functions perform operations on arithmetic, string, or date/time expressions, including the return values of other functions.

List of functions

The following miscellaneous functions are available:

- "ARGN function [Miscellaneous]" on page 104
- "COALESCE function [Miscellaneous]" on page 118
- "COMPRESS function [String]" on page 121
- "CONFLICT function [Miscellaneous]" on page 123
- "DECOMPRESS function [String]" on page 148
- "DECRYPT function [String]" on page 149
- "ENCRYPT function [String]" on page 154
- "ERRORMSG function [Miscellaneous]" on page 155
- "ESTIMATE function [Miscellaneous]" on page 156
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 156
- "EXPERIENCE_ESTIMATE function [Miscellaneous]" on page 162
- "EXPLANATION function [Miscellaneous]" on page 163
- "EXPRTYPE function [Miscellaneous]" on page 164
- "GET_IDENTITY function [Miscellaneous]" on page 168
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 169
- "GREATER function [Miscellaneous]" on page 171
- "IDENTITY function [Miscellaneous]" on page 182
- "IFNULL function [Miscellaneous]" on page 182
- "INDEX_ESTIMATE function [Miscellaneous]" on page 183
- "ISNULL function [Miscellaneous]" on page 186
- "LESSER function [Miscellaneous]" on page 191
- "NEWID function [Miscellaneous]" on page 204
- "NULLIF function [Miscellaneous]" on page 210
- "NUMBER function [Miscellaneous]" on page 211
- "PLAN function [Miscellaneous]" on page 214
- "REWRITE function [Miscellaneous]" on page 236

- "ROW_NUMBER function [Miscellaneous]" on page 240
- "SQLDIALECT function [Miscellaneous]" on page 255
- "SQLFLAGGER function [Miscellaneous]" on page 255
- "TRACEBACK function [Miscellaneous]" on page 268
- "TRANSACTSQL function [Miscellaneous]" on page 269
- "VAREXISTS function [Miscellaneous]" on page 278
- "WATCOMSQL function [Miscellaneous]" on page 278

Numeric functions

Numeric functions perform mathematical operations on numerical data types or return numeric information.

List of functions

The following numeric functions are available:

- "ABS function [Numeric]" on page 103
- "ACOS function [Numeric]" on page 103
- "ASIN function [Numeric]" on page 105
- "ATAN function [Numeric]" on page 106
- "ATAN2 function [Numeric]" on page 106
- "CEILING function [Numeric]" on page 115
- "COS function [Numeric]" on page 128
- "COT function [Numeric]" on page 129
- "DEGREES function [Numeric]" on page 150
- "EXP function [Numeric]" on page 162
- "FLOOR function [Numeric]" on page 167
- "LOG function [Numeric]" on page 195
- "LOG10 function [Numeric]" on page 196
- "MOD function [Numeric]" on page 201
- "PI function [Numeric]" on page 214
- "POWER function [Numeric]" on page 215
- "RADIANS function [Numeric]" on page 219
- "RAND function [Numeric]" on page 219
- "REMAINDER function [Numeric]" on page 233
- "ROUND function [Numeric]" on page 239
- "SIGN function [Numeric]" on page 246
- "SIN function [Numeric]" on page 248
- "SQRT function [Numeric]" on page 256
- "TAN function [Numeric]" on page 265
- "TRUNCNUM function [Numeric]" on page 270

HTTP and SOAP functions

HTTP functions facilitate the handling of HTTP requests within web services. Likewise, SOAP functions facilitate the handling of SOAP requests within web services.

List of functions

The following HTTP functions are available:

- "HTML_DECODE function [Miscellaneous]" on page 176
- "HTML_ENCODE function [Miscellaneous]" on page 177
- "HTTP_DECODE function [HTTP]" on page 178
- "HTTP_ENCODE function [HTTP]" on page 178
- "HTTP_HEADER function [HTTP]" on page 179
- "HTTP_VARIABLE function [HTTP]" on page 181
- "NEXT_HTTP_HEADER function [HTTP]" on page 207
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 208

The following SOAP functions are available:

- "NEXT_SOAP_HEADER function [SOAP]" on page 209
- "SOAP_HEADER function [SOAP]" on page 248

String functions

String functions perform conversion, extraction, or manipulation operations on strings, or return information about strings.

When working in a multibyte character set, check carefully whether the function being used returns information concerning characters or bytes.

List of functions

The following string functions are available:

- "ASCII function [String]" on page 104
- "BASE64_DECODE function [String]" on page 108
- "BASE64_ENCODE function [String]" on page 108
- "BYTE_LENGTH function [String]" on page 113
- "BYTE_SUBSTR function [String]" on page 114
- "CHAR function [String]" on page 116
- "CHARINDEX function [String]" on page 117
- "CHAR_LENGTH function [String]" on page 118
- "COMPARE function [String]" on page 119
- "COMPRESS function [String]" on page 121
- "CSCONVERT function [String]" on page 133
- "DECOMPRESS function [String]" on page 148
- "DECRYPT function [String]" on page 149
- "DIFFERENCE function [String]" on page 152
- "ENCRYPT function [String]" on page 154
- "HASH function [String]" on page 172
- "INSERTSTR function [String]" on page 184
- "LCASE function [String]" on page 189

- "LEFT function [String]" on page 190
- "LENGTH function [String]" on page 190
- "LOCATE function [String]" on page 194
- "LOWER function [String]" on page 196
- "LTRIM function [String]" on page 197
- "PATINDEX function [String]" on page 212
- "REPEAT function [String]" on page 233
- "REPLACE function [String]" on page 234
- "REPLICATE function [String]" on page 235
- "REVERSE function [String]" on page 236
- "RIGHT function [String]" on page 238
- "RTRIM function [String]" on page 242
- "SIMILAR function [String]" on page 247
- "SORTKEY function [String]" on page 249
- "SOUNDEX function [String]" on page 253
- "SPACE function [String]" on page 254
- "STR function [String]" on page 259
- "STRING function [String]" on page 260
- "STRTOUUID function [String]" on page 261
- "STUFF function [String]" on page 262
- "SUBSTRING function [String]" on page 262
- "TO_CHAR function [String]" on page 266
- "TO_NCHAR function [String]" on page 267
- "TRIM function [String]" on page 270
- "UCASE function [String]" on page 271
- "UNICODE function [String]" on page 272
- "UNISTR function [String]" on page 272
- "UPPER function [String]" on page 273
- "UUIDTOSTR function [String]" on page 274
- "XMLCONCAT function [String]" on page 281
- "XMLELEMENT function [String]" on page 282
- "XMLFOREST function [String]" on page 284
- "XMLGEN function [String]" on page 285

System functions

System functions return system information.

List of functions

The following system functions are available:

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121
- "CONNECTION_PROPERTY function [System]" on page 122
- "DATALENGTH function [System]" on page 136
- "DB_ID function [System]" on page 146
- "DB_NAME function [System]" on page 147

- "DB_EXTENDED_PROPERTY function [System]" on page 143
- "DB_PROPERTY function [System]" on page 147
- "EVENT_CONDITION function [System]" on page 158
- "EVENT_CONDITION_NAME function [System]" on page 159
- "EVENT_PARAMETER function [System]" on page 160
- "NEXT_CONNECTION function [System]" on page 205
- "NEXT_DATABASE function [System]" on page 207
- "PROPERTY function [System]" on page 216
- "PROPERTY_DESCRIPTION function [System]" on page 217
- "PROPERTY_NAME function [System]" on page 217
- "PROPERTY_NUMBER function [System]" on page 218

Notes

- Some of the system functions are implemented in SQL Anywhere as stored procedures.
- The db_id, db_name, and datalength functions are implemented as built-in functions.

The implemented system functions are described in the following table.

System function	Description
col_length(*table-name*, *column-name* **)**	Returns the defined length of column
col_name(*table-id*, *column-id* [, *database-id*] **)**	Returns the column name
datalength(*expression* **)**	Returns the length of the expression, in bytes
db_id([*database-name*] **)**	Returns the database ID number
db_name([*database-id*] **)**	Returns the database name
index_col (*table-name*, *index-id*, *key_#* [, *userid*] **)**	Returns the name of the indexed column
object_id (*object-name* **)**	Returns the object ID
object_name (*object-id* [, *database-id*] **)**	Returns the object name
suser_id([*user-name*] **)**	Returns an integer user identification number
suser_name([*user-id*] **)**	Returns the user ID
tsequal (*timestamp*, *timestamp2* **)**	In SQL Anywhere, tsequal compares timestamp values (truncated to milliseconds) to prevent an update on a row that has been modified since it was selected. When timestamps are different, false (0) is returned. See "Using tsequal for updates" [*SQL Anywhere Server - SQL Usage*]. The use of tsequal is deprecated.
user_id([*user-name*] **)**	Returns an integer user identification number. This does not return the SQL Anywhere user ID.

System function	Description
user_name([*user-id*] **)**	Returns the user ID

Text and image functions

Text and image functions operate on text and image data types. SQL Anywhere supports only the textptr text and image function.

List of functions

The following text and image function is available:

- "TEXTPTR function [Text and image]" on page 265

Alphabetical list of functions

Each function is listed, and the function type (numeric, character, and so on) is indicated next to it.

For links to all functions of a given type, see "Function types" on page 93.

ABS function [Numeric]

Returns the absolute value of a numeric expression.

Syntax

ABS(*numeric-expression* **)**

Parameters

numeric expression The number whose absolute value is to be returned.

Standards and compatibility

- **SQL/2003** SQL foundation feature outside of core SQL.

Example

The following statement returns the value 66.

```
SELECT ABS( -66 );
```

ACOS function [Numeric]

Returns the arc-cosine, in radians, of a numeric expression.

Syntax

ACOS(*numeric-expression* **)**

Parameters

numeric-expression The cosine of the angle.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ASIN function [Numeric]" on page 105
- "ATAN function [Numeric]" on page 106
- "ATAN2 function [Numeric]" on page 106
- "COS function [Numeric]" on page 128

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-cosine value for 0.52.

```
SELECT ACOS( 0.52 );
```

ARGN function [Miscellaneous]

Returns a selected argument from a list of arguments.

Syntax

ARGN(*integer-expression*, *expression* [, …] **)**

Parameters

integer-expression The position of an argument within the list of expressions.

expression An expression of any data type passed into the function. All supplied expressions must be of the same data type.

Remarks

Using the value of the *integer-expression* as n, returns the nth argument (starting at 1) from the remaining list of arguments. While the expressions can be of any data type, they must all be of the same data type. The integer expression must be from one to the number of expressions in the list or NULL is returned. Multiple expressions are separated by a comma.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 6.

```
SELECT ARGN( 6, 1,2,3,4,5,6 );
```

ASCII function [String]

Returns the integer ASCII value of the first byte in a string-expression.

Syntax

ASCII(*string-expression* **)**

Parameters

string-expression The string.

Remarks

If the string is empty, then ASCII returns zero. Literal strings must be enclosed in quotes. If the database character set is multibyte and the first character of the parameter string consists of more than one byte, the result is NULL.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 90.

```
SELECT ASCII( 'Z' );
```

ASIN function [Numeric]

Returns the arc-sine, in radians, of a number.

Syntax

ASIN(*numeric-expression* **)**

Parameters

numeric-expression The sine of the angle.

Remarks

The SIN and ASIN functions are inverse operations.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ACOS function [Numeric]" on page 103
- "ATAN function [Numeric]" on page 106
- "ATAN2 function [Numeric]" on page 106
- "SIN function [Numeric]" on page 248

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-sine value for 0.52.

```
SELECT ASIN( 0.52 );
```


ATAN function [Numeric]

Returns the arc-tangent, in radians, of a number.

Syntax

ATAN(*numeric-expression* **)**

Remarks

The ATAN and TAN functions are inverse operations.

Parameters

numeric-expression The tangent of the angle.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ACOS function [Numeric]" on page 103
- "ASIN function [Numeric]" on page 105
- "ATAN2 function [Numeric]" on page 106
- "TAN function [Numeric]" on page 265

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-tangent value for 0.52.

```
SELECT ATAN( 0.52 );
```

ATAN2 function [Numeric]

Returns the arc-tangent, in radians, of the ratio of two numbers.

Syntax

{ **ATN2** | **ATAN2** }**(** *numeric-expression-1*, *numeric-expression-2* **)**

Parameters

numeric-expression-1 The numerator in the ratio whose arc-tangent is calculated.

numeric-expression-2 The denominator in the ratio whose arc-tangent is calculated.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ACOS function [Numeric]" on page 103
- "ASIN function [Numeric]" on page 105
- "ATAN function [Numeric]" on page 106
- "TAN function [Numeric]" on page 265

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-tangent value for the ratio 0.52 to 0.60.

```
SELECT ATAN2( 0.52, 0.60 );
```

AVG function [Aggregate]

Computes the average, for a set of rows, of a numeric expression or of a set unique values.

Syntax 1

AVG(*numeric-expression* | **DISTINCT** *numeric-expression* **)**

Syntax 2

AVG(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

numeric-expression The expression whose average is calculated over a set of rows.

DISTINCT numeric-expression Computes the average of the unique numeric values in the input.

Remarks

This average does not include rows where the *numeric-expression* is the NULL value. Returns the NULL value for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "SUM function [Aggregate]" on page 264
- "COUNT function [Aggregate]" on page 129

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 49988.623200.

```
SELECT AVG( Salary ) FROM Employees ;
```

The following statement could be used to determine the average based on unique prices in the production list:

```
SELECT AVG( DISTINCT ListPrice ) FROM Production ;
```

BASE64_DECODE function [String]

Decodes data using the MIME base64 format and returns the string as a LONG VARCHAR.

Syntax

BASE64_DECODE(*string-expression* **)**

Parameters

string-expression The string that is to be decoded. Note that the string must be base64-encoded.

See also

- "BASE64_ENCODE function [String]" on page 108
- "String functions" on page 99

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following inserts an image into an image table from an embedded SQL program. The input data (host variable) must be base64 encoded.

```
EXEC SQL INSERT INTO images ( image_data ) VALUES ( BASE64_DECODE ( :img ) );
```

BASE64_ENCODE function [String]

Encodes data using the MIME base64 format and returns it as a 7-bit ASCII string.

Syntax

BASE64_ENCODE(*string-expression* **)**

Parameters

string-expression The string that is to be encoded.

See also

- "BASE64_DECODE function [String]" on page 108
- "String functions" on page 99

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following retrieves data from a table containing images and returns it in ASCII format. The resulting string can be embedded into an email message, and then decoded by the recipient to retrieve the original image.

```
SELECT BASE64_ENCODE( image_data ) FROM IMAGES ;
```

BIT_LENGTH function [Bit array]

Returns the number of bits stored in the array.

Syntax

BIT_LENGTH(*bit-expression* **)**

Parameters

bit-expression The bit expression for which the length is to be determined.

See also

- "BIT_LENGTH function [Bit array]" on page 109
- "CHAR_LENGTH function [String]" on page 118

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following statement returns the value 8:

```
SELECT BIT_LENGTH( '01101011' );
```

BIT_SUBSTR function [Bit array]

Returns a sub-array of a bit array.

Syntax

BIT_SUBSTR(*bit-expression* [**,** *start* [**,** *length*]] **)**

Parameters

bit-expression The bit array from which the sub-array is to be extracted.

start The start position of the sub-array to return. A negative starting position specifies the number of bits from the end of the array instead of the beginning. The first bit in the array is at position 1.

length The length of the sub-array to return. A positive length specifies that the sub-array ends *length* bits to the right of the starting position, while a negative length returns, at most, *length* bits up to, and including, the starting position, from the left of the starting position.

Remarks

Both *start* and *length* can be either positive or negative. Using appropriate combinations of negative and positive numbers, you can get a sub-array from either the beginning or end of the string. Using a negative number for *length* does not impact the order of the bits returned in the sub-array.

If *length* is specified, the sub-array is restricted to that length. If start is zero and length is non-negative, a start value of 1 is used. If start is zero and length is negative, a start value of -1 is used.

If *length* is not specified, selection continues to the end of the array.

The BIT_SUBSTR function is equivalent to, but faster than, the following:

```
CAST( SUBSTR( CAST( bit-expression AS VARCHAR),
start [, length ] )
AS VARBIT )
```

See also

- "SUBSTRING function [String]" on page 262

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following statement returns 1101:

```
SELECT BIT_SUBSTR( '001101', 3 );
```

The following statement returns 10110:

```
SELECT BIT_SUBSTR( '01011011101111011111', 2, 5 );
```

The following statement returns 11111:

```
SELECT BIT_SUBSTR( '01011011101111011111', -5, 5 );
```

BIT_AND function [Aggregate]

Takes *n* bit arrays and returns a bitwise AND-ing of its arguments using the following logic: for each bit compared, if all bits are 1, return 1; otherwise, return 0.

Syntax

BIT_AND(*bit-expression* **)**

Parameters

expression The expression for which the value is to be determined. This is commonly a column name.

See also

- "BIT_OR function [Aggregate]" on page 111
- "BIT_XOR function [Aggregate]" on page 112
- "Bitwise operators" on page 13

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you have the following table, t, containing a single column, a, which is a VARBIT data type.

a
0001
0111
0100
0011

You want to know the AND value for the column. You enter the following SELECT statement, which returns 0000:

```
SELECT BIT_AND( a ) FROM t;
```

This result is determined as follows:

1. Row 1 (0001) is compared with Row 2 (0111), and results in 0001 (both values had a 1 in the fourth bit).
2. The result from the previous comparison (0001) is compared with Row 3 (0100), and results in 0000 (neither value had a 1 in the same bit).
3. The result from the previous comparison (0000) is compared with Row 4 (0011), and results in 0000 (neither value had a 1 in the same bit).

BIT_OR function [Aggregate]

Takes *n* bit arrays and returns a bitwise OR-ing of its arguments using the following logic: for each bit compared, if any bit is 1, return 1; otherwise, return 0.

Syntax

BIT_OR(*bit-expression* **)**

Parameters

expression The expression for which the value is to be determined. This is commonly a column name.

See also

- "BIT_AND function [Aggregate]" on page 110
- "BIT_XOR function [Aggregate]" on page 112
- "Bitwise operators" on page 13

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you have the following table, t, containing a single column, a, which is a VARBIT data type.

a
0001
0111
0100
0011

You want to know the OR value for the column. You enter the following SELECT statement, which returns 0111:

```
SELECT BIT_OR( a ) FROM t;
```

This result is determined as follows:

1. Row 1 (0001) is compared with Row 2 (0111), and results in 0111.
2. The result from the previous comparison (0111) is compared with Row 3 (0100), and results in 0111.
3. The result from the previous comparison (0111) is compared with Row 4 (0011), and results in 0111.

BIT_XOR function [Aggregate]

Takes *n* bit arrays and returns a bitwise exclusive OR-ing of its arguments using the following logic: for each bit compared, if there are an odd number of arguments with set bits (odd parity), return 1; otherwise, return 0.

Syntax

BIT_XOR(*bit-expression* **)**

Parameters

expression The expression for which the value is to be determined. This is commonly a column name.

See also

- "BIT_AND function [Aggregate]" on page 110
- "BIT_OR function [Aggregate]" on page 111

- "Bitwise operators" on page 13

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you have the following table, t, containing a single column, a, which is a VARBIT data type.

a
0001
0111
0100
0011

You want to know the XOR value for the column. You enter the following SELECT statement, which returns 0001:

```
SELECT BIT_XOR( a ) FROM t;
```

This result is determined as follows:

1. Row 1 (0001) is compared with Row 2 (0111), and results in 0110.
2. The result from the previous comparison (0110) is compared with Row 3 (0100), and results in 0010.
3. The result from the previous comparison (0010) is compared with Row 4 (0011), and results in 0001.

BYTE_LENGTH function [String]

Returns the number of bytes in a string.

Syntax

BYTE_LENGTH(*string-expression* **)**

Parameters

string-expression The string whose length is to be calculated.

Remarks

Trailing white space characters in the *string-expression* are included in the length returned.

The return value of a NULL string is NULL.

If the string is in a multibyte character set, the BYTE_LENGTH value may differ from the number of characters returned by CHAR_LENGTH.

This function supports NCHAR inputs and/or outputs.

See also

- "CHAR_LENGTH function [String]" on page 118
- "DATALENGTH function [System]" on page 136
- "LENGTH function [String]" on page 190
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 12.

```
SELECT BYTE_LENGTH( 'Test Message' );
```

BYTE_SUBSTR function [String]

Returns a substring of a string. The substring is calculated using bytes, not characters.

Syntax

BYTE_SUBSTR(*string-expression*, *start* [, *length*] **)**

Parameters

string-expression The string from which the substring is taken.

start An integer expression indicating the start of the substring. A positive integer starts from the beginning of the string, with the first character being position 1. A negative integer specifies a substring starting from the end of the string, the final character being at position -1.

length An integer expression indicating the length of the substring. A positive *length* specifies the number of bytes to be taken *starting* at the start position. A negative *length* returns at most *length* bytes up to, and including, the starting position, from the left of the starting position.

Remarks

If *length* is specified, the substring is restricted to that number of bytes. Both *start* and *length* can be either positive or negative. Using appropriate combinations of negative and positive numbers, you can get a substring from either the beginning or end of the string.

If *start* is zero and length is non-negative, a *start* value of 1 is used. If *start* is zero and *length* is negative, a start value of -1 is used.

This function supports NCHAR inputs and/or outputs.

See also

- "SUBSTRING function [String]" on page 262
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Test.

```
SELECT BYTE_SUBSTR( 'Test Message', 1, 4 );
```

CAST function [Data type conversion]

Returns the value of an expression converted to a supplied data type.

Syntax

CAST(*expression* **AS** *datatype* **)**

Parameters

expression The expression to be converted.

data type The target data type.

Remarks

If you do not indicate a length for character string types, the database server chooses an appropriate length. If neither precision nor scale is specified for a DECIMAL conversion, the database server selects appropriate values.

If you use the CAST function to truncate strings, the string_rtruncation database option must be set to OFF; otherwise, there will be an error. It is recommended that you use the LEFT function to truncate strings.

See also

- "CONVERT function [Data type conversion]" on page 125
- "LEFT function [String]" on page 190

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following function ensures a string is used as a date:

```
SELECT CAST( '2000-10-31' AS DATE );
```

The value of the expression `1 + 2` is calculated, and the result is then cast into a single-character string.

```
SELECT CAST( 1 + 2 AS CHAR );
```

CEILING function [Numeric]

Returns the ceiling of a number.

Syntax

CEILING(*numeric-expression* **)**

Parameters

numeric-expression The number whose ceiling is to be calculated.

Remarks

The Ceiling function returns the first integer that is greater or equal to a given value. For positive numbers, this is also known as "rounding up."

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "FLOOR function [Numeric]" on page 167

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 60.

```
SELECT CEILING( 59.84567 );
```

CHAR function [String]

Returns the character with the ASCII value of a number.

Syntax

CHAR(*integer-expression* **)**

Parameters

integer-expression The number to be converted to an ASCII character. The number must be in the range 0 to 255, inclusive.

Remarks

The character returned corresponds to the supplied numeric expression in the current database character set, according to a binary sort order.

CHAR returns NULL for integer expressions with values greater than 255 or less than zero.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Y.


```
SELECT CHAR( 89 );
```

CHARINDEX function [String]

Returns the position of one string in another.

Syntax

CHARINDEX(*string-expression-1*, *string-expression-2* **)**

Parameters

string-expression-1 The string for which you are searching.

string-expression-2 The string to be searched.

Remarks

The first character of *string-expression-1* is identified as 1. If the string being searched contains more than one instance of the other string, then the CHARINDEX function returns the position of the first instance.

If the string being searched does not contain the other string, then the CHARINDEX function returns 0.

This function supports NCHAR inputs and/or outputs.

See also

- "SUBSTRING function [String]" on page 262
- "REPLACE function [String]" on page 234
- "LOCATE function [String]" on page 194
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns last and first names from the Surname and GivenName tables, but only when the last name includes the letter K:

```
SELECT Surname, GivenName
FROM Employees
WHERE CHARINDEX( 'K', Surname ) = 1 ;
```

Results returned:

Surname	GivenName
Klobucher	James
Kuo	Felicia
Kelly	Moira

CHAR_LENGTH function [String]

Returns the number of characters in a string.

Syntax

CHAR_LENGTH (*string-expression* **)**

Parameters

string-expression The string whose length is to be calculated.

Remarks

Trailing white space characters are included in the length returned.

The return value of a NULL string is NULL.

If the string is in a multibyte character set, the value returned by the CHAR_LENGTH function may differ from the number of bytes returned by the BYTE_LENGTH function.

> **Note**
> You can use the CHAR_LENGTH function and the LENGTH function interchangeably for CHAR, VARCHAR, LONG VARCHAR, and NCHAR data types. However, you must use the LENGTH function for BINARY and bit array data types.

This function supports NCHAR inputs and/or outputs.

See also

- "BYTE_LENGTH function [String]" on page 113
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value 8.

```
SELECT CHAR_LENGTH( 'Chemical' );
```

COALESCE function [Miscellaneous]

Returns the first non-NULL expression from a list. This function is identical to the ISNULL function.

Syntax

COALESCE(*expression*, *expression* [, …] **)**

Parameters

expression Any expression.

At least two expressions must be passed into the function, and all expressions must be comparable.

Remarks

The result is NULL only if all the arguments are NULL.

The parameters can be of any scalar type, but not necessarily same type.

For a more detailed description of how the database server processes this function, see "ISNULL function [Miscellaneous]" on page 186.

See also

- "ISNULL function [Miscellaneous]" on page 186

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value 34.

```
SELECT COALESCE( NULL, 34, 13, 0 );
```

COMPARE function [String]

Allows you to compare two character strings based on alternate collation rules.

Syntax

COMPARE(
string-expression-1,
string-expression-2
[**,** { *collation-id*
| *collation-name*[**(***collation-tailoring-string***)**] }]
)

Parameters

string-expression-1 The first string expression.

string-expression-2 The second string expression.

The string expression can only contain characters that are encoded in the database's character set.

collation-id A variable or integer constant that specifies the sort order to use. You can only use a *collation-id* for built-in collations. See "SORTKEY function [String]" on page 249.

If you do not specify a collation name or ID, the default is Default Unicode multilingual.

collation-name A string or a character variable that specifies the name of the collation to use. You can also specify char_collation or db_collation (for example, `COMPARE( 'abc', 'ABC', 'char_collation' );`) to use the database's CHAR collation. Similarly, you can specify nchar_collation to use the database's NCHAR collation. For a list of valid collation names, see "SORTKEY function [String]" on page 249.

collation-tailoring-string Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the character comparison. These options take the form of keyword=value pairs in parentheses, following the collation name. For example, `'UCA (locale=es;case=LowerFirst;accent=respect)'`. The syntax for specifying these options is identical to the syntax defined for the COLLATION clause of the CREATE DATABASE statement. See "Collation tailoring options" on page 376.

Note
All of the collation tailoring options are supported when specifying the UCA collation. For all other collations, only case sensitivity tailoring option is supported.

Remarks

The COMPARE function returns the following values, based on the collation rules that you choose:

Value	Meaning
1	*string-expression-1* is greater than *string-expression-2*
0	*string-expression-1* is equal to *string-expression-2*
-1	*string-expression-1* is less than *string-expression-2*

The COMPARE function does not equate empty strings and strings containing only spaces, even if the database has blank-padding enabled. The COMPARE function uses the SORTKEY function to generate collation keys for comparison. Therefore, an empty string, a string with one space, and a string with two spaces do not compare equally.

If either *string-expression-1* or *string-expression-2* is NULL, the result is NULL.

See also

- "SORTKEY function [String]" on page 249
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example performs three comparisons using the COMPARE function:

```
SELECT COMPARE( 'abc','ABC','UCA(case=LowerFirst)' ),
       COMPARE( 'abc','ABC','UCA(case=Ignore)' ),
       COMPARE( 'abc','ABC','UCA(case=UpperFirst)' );
```

The values returned are -1, 0, 1, indicating the result of each comparison. The first comparison results in -1, indicating that *string-expression-2* ('ABC') is less than *string-expresssion-1* ('abc'). This is because case sensitivity is set to LowerFirst in the first COMPARE statement.

COMPRESS function [String]

Compresses the string and returns a value of type LONG BINARY.

Syntax

COMPRESS(*string-expression* [**,** '*compression-algorithm-alias*']**)**

Parameters

string-expression The string to be compressed. Binary values can be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

compression-algorithm-alias Alias for the algorithm to use for compression. The supported values are zip and gzip (both are based on the same algorithm, but use different headers and trailers).

Zip is a widely supported compression algorithm. Gzip is compatible with the gzip utility on Unix, whereas the zip algorithm is not.

Decompression must be performed with the same algorithm.

For more information, see "DECOMPRESS function [String]" on page 148.

Remarks

The COMPRESS function returns a LONG BINARY value that is usually shorter than the binary string passed to the function. This value is not human-readable. If the value returned is longer than the original string, its maximum size will not be larger than a 0.1% increase over the original string + 12 bytes. You can decompress a compressed *string-expression* using the DECOMPRESS function.

If you are storing compressed values in a table, the column should be BINARY or LONG BINARY so that character set conversion is not performed on the data.

See also

- "DECOMPRESS function [String]" on page 148
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the length of the binary string created by compressing the string 'Hello World' using the gzip algorithm. This example can be useful when you want to determine whether a value has a shorter length when compressed.

```
SELECT LENGTH( COMPRESS( 'Hello world', 'gzip' ) );
```

CONNECTION_EXTENDED_PROPERTY function [String]

Returns the value of the given property. Allows an optional property-specific string parameter to be specified.

Syntax

CONNECTION_EXTENDED_PROPERTY(
{ *property-id* | *property-name* }
[, *property-specific-argument*]
)

Parameters

property-id The connection property ID.

property-name The connection property name. Possible property names are CharSet and NcharCharSet.

property-specific-argument Optional property-specific string parameter associated with the following connection properties.

- **CharSet** Returns the CHAR character set label for the connection as it is known by the specified standard. The possible values include: ASE, IANA, MIME, JAVA, WINDOWS, UTR22, IBM, and ICU. The default is IANA unless the database connection was made through TDS in which case ASE is the default.
- **NcharCharSet** Returns the NCHAR character set label for the connection as it is known by the specified standard. The possible values are the same as listed above for CharSet.

Remarks

The CONNECTION_EXTENDED_PROPERTY function returns extended connection properties. The returned value is a LONG VARCHAR, and applies to the current connection.

The CONNECTION_EXTENDED_PROPERTY function is similar to the CONNECTION_PROPERTY function except that it allows an optional property-specific string parameter to be specified. The interpretation of the property-specific argument depends on the property ID or name specified in the first argument.

You can use the CONNECTION_EXTENDED_PROPERTY function to return the value for any connection property. However, extended information is only available for the extended properties.

See also

- "Connection-level properties" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_PROPERTY function [System]" on page 122
- "DB_EXTENDED_PROPERTY function [System]" on page 143
- "DB_PROPERTY function [System]" on page 147

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the CHAR character set of the current connection as it is known by the Java standard:

```
SELECT CONNECTION_EXTENDED_PROPERTY( 'charset', 'Java' );
```

CONNECTION_PROPERTY function [System]

Returns the value of a given connection property as a string.

Syntax

CONNECTION_PROPERTY(
{ *integer-expression-1* | *string-expression* }
[, *integer-expression-2*] **)**

Parameters

integer-expression-1 In most cases it is more convenient to supply a string expression as the first argument. If you do supply an integer-expression, it is the connection property ID. You can determine this using the PROPERTY_NUMBER function.

string-expression The connection property Name. Either the property ID or the property name must be specified.

For a list of connection properties, see "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

integer-expression-2 The connection ID of the current database connection. The current connection is used if this argument is omitted.

Remarks

The current connection is used if the second argument is omitted.

See also

- "Connection-level properties" [*SQL Anywhere Server - Database Administration*]
- "PROPERTY_NUMBER function [System]" on page 218

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the number of prepared statements being maintained.

```
SELECT CONNECTION_PROPERTY( 'PrepStmt' );
```

CONFLICT function [Miscellaneous]

Indicates if a column is a source of conflict for an UPDATE being performed against a consolidated database in a SQL Remote environment.

Syntax

CONFLICT(*column-name* **)**

Parameters

column-name The name of the column being tested for conflicts.

Remarks

Returns TRUE if the column appears in the VERIFY list of an UPDATE statement executed by the SQL Remote Message Agent and if the value provided in the VALUES list of that statement does not match the original value of the column in the row being updated. Otherwise, returns FALSE.

See also

- "CREATE TRIGGER statement" on page 462
- "Managing conflicts" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The CONFLICT function is intended for use in SQL Remote RESOLVE UPDATE triggers to avoid error messages. To illustrate the use of the CONFLICT function, consider the following table:

```
CREATE TABLE Admin (
  PKey bigint NOT NULL DEFAULT GLOBAL AUTOINCREMENT,
  TextCol CHAR(20) NULL, PRIMARY KEY ( PKey ) );
```

Assume that consolidated and remote databases both have the following row in the Admin table:

```
1, 'Initial'
```

Now, at the consolidated database, update the row as follows:

```
UPDATE Admin SET TextCol = 'Consolidated Update' WHERE PKey = 1;
```

At the remote database, update the row to a different value as follows:

```
UPDATE Admin SET TextCol = 'Remote Update' WHERE PKey = 1;
```

Next, run dbremote on the remote database. It generates a message file with the following statements in it, to be executed at the consolidated database:

```
UPDATE Admin SET TextCol='Remote Update',
VERIFY ( TextCol )
VALUES ( 'Initial' )
WHERE PKey=1;
```

When the SQL Remote Message Agent runs at the consolidated database and applies this UPDATE statement, SQL Anywhere uses the VERIFY and VALUES clause to determine whether a RESOLVE UPDATE trigger will fire. A RESOLVE UPDATE trigger fires only when the update is executed from the SQL Remote Message Agent against a consolidated database. Here is a RESOLVE UPDATE trigger:

```
CREATE TRIGGER ResolveUpdateAdmin
RESOLVE UPDATE ON Admin
REFERENCING OLD AS OldConsolidated
    NEW AS NewRemote
    REMOTE as OldRemote
FOR EACH ROW BEGIN
  MESSAGE 'OLD';
  MESSAGE OldConsolidated.PKey || ',' || OldConsolidated.TextCol;
  MESSAGE 'NEW';
  MESSAGE NewRemote.PKey || ',' || NewRemote.TextCol;
```



```
  MESSAGE 'REMOTE';
  MESSAGE OldRemote.PKey || ',' || OldRemote.TextCol;
END;
```

The RESOLVE UPDATE trigger fires because the current value of the TextCol column at the consolidated database (`'Consolidated Update'`) does not match the value in the VALUES clause for the associated column (`'Initial'`).

This trigger results in a failure because the PKey column was not modified in the UPDATE statement executed on the remote, so there is no OldRemote.PKey value accessible from this trigger.

The CONFLICT function helps to avoid this error by returning the following values:

- If there is no OldRemote.PKey value, return FALSE.
- If there is an OldRemote.PKey value, but it matches OldConsolidated.PKey, return FALSE.
- If there is an OldRemote.PKey value, and it is different than OldConsolidated.PKey, return TRUE.

You can use the CONFLICT function to rewrite the trigger as follows and avoid the error:

```
CREATE TRIGGER ResolveUpdateAdmin
RESOLVE UPDATE ON Admin
REFERENCING OLD AS OldConsolidated
    NEW AS NewRemote
    REMOTE as OldRemote
FOR EACH ROW BEGIN
  message 'OLD';
  message OldConsolidated.PKey || ',' || OldConsolidated.TextCol;
  message 'NEW';
  message NewRemote.PKey || ',' || NewRemote.TextCol;
  message 'REMOTE';
  if CONFLICT( PKey ) then
    message OldRemote.PKey;
  end if;
  if CONFLICT( TextCol ) then
    message OldRemote.TextCol;
  end if;
END;
```

CONVERT function [Data type conversion]

Returns an expression converted to a supplied data type.

Syntax

CONVERT(*datatype*, *expression* [, *format-style*] **)**

Parameters

datatype The data type to which the expression is converted.

expression The expression to be converted.

format-style The style code to apply to the outputted value. Use this parameter when converting strings to date or time data types, and vice versa. The table below shows the supported style codes, followed by a

representation of the output format produced by that style code. The style codes are separated into two columns, depending on whether the century is included in the output format (for example, 06 versus 2006).

Without century (yy) style codes	With century (yyyy) style codes	Output format
-	0 or 100	Mmm dd yyyy hh:nnAA
1	101	mm/dd/yy[yy]
2	102	[yy]yy.mm.dd
3	103	dd/mm/yy[yy]
4	104	dd.mm.yy[yy]
5	105	dd-mm-yy[yy]
6	106	dd Mmm yy[yy]
7	107	Mmm dd, yy[yy]
8	108	hh:nn:ss
-	9 or 109	Mmm dd yyyy hh:nn:ss:sssAA
10	110	mm-dd-yy[yy]
11	111	[yy]yy/mm/dd
12	112	[yy]yymmdd
-	13 or 113	dd Mmm yyyy hh:nn:ss:sss (24 hour clock, Europe default + milliseconds, 4-digit year)
-	14 or 114	hh:nn:ss:sss (24 hour clock)
-	20 or 120	yyyy-mm-dd hh:nn:ss (24-hour clock, ODBC canonical, 4-digit year)
-	21 or 121	yyyy-mm-dd hh:nn:ss.sss (24 hour clock, ODBC canonical with mil-liseconds, 4-digit year)

Remarks

If no *format-style* argument is provided, style code 0 is used.

For a description of the styles produced by each output symbol (such as Mmm), see "date_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

See also

- "CAST function [Data type conversion]" on page 115

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements illustrate the use of format style.

```
SELECT CONVERT( CHAR( 20 ), OrderDate, 104 ) FROM SalesOrders ;
```

OrderDate
16.03.2000
20.03.2000
23.03.2000
25.03.2000
…

```
SELECT CONVERT( CHAR( 20 ), OrderDate, 7 ) FROM SalesOrders;
```

OrderDate
Mar 16, 00
Mar 20, 00
Mar 23, 00
Mar 25, 00
…

The following statement illustrates conversion to an integer, and returns the value 5.

```
SELECT CONVERT( integer, 5.2 );
```

CORR function [Aggregate]

Returns the correlation coefficient of a set of number pairs.

Syntax

CORR(*dependent-expression*, *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) after eliminating the pairs for which either *dependent-expression* or *independent-expression* is NULL. The following computation is made:

COVAR_POP (*x*, *y*) / **STDDEV_POP** (*x*) * **STDDEV_POP** (*y*)

where *x* represents the *dependent-expression* and *y* represents the *independent-expression*.

See also

- "Aggregate functions" on page 93
- "COVAR_POP function [Aggregate]" on page 131
- "STDDEV_POP function [Aggregate]" on page 257

Standards and compatibility

- **SQL/2003** SQL foundation feature outside of core SQL.

Example

The following example performs a correlation to discover whether age is associated with income level. This function returns the value 0.4402267564599596.

```
SELECT CORR( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) ) FROM Employees;
```

COS function [Numeric]

Converts a number from radians to cosine.

Syntax

COS(*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Remarks

The COS function returns the cosine of the angle given by *numeric-expression*.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

See also

- "ACOS function [Numeric]" on page 103
- "COT function [Numeric]" on page 129
- "SIN function [Numeric]" on page 248
- "TAN function [Numeric]" on page 265

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value of the cosine of an angle 0.52 radians.

```
SELECT COS( 0.52 );
```

COT function [Numeric]

Converts a number from radians to cotangent.

Syntax

COT(*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Remarks

The COT function returns the cotangent of the angle given by *numeric-expression*.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

See also

- "COS function [Numeric]" on page 128
- "SIN function [Numeric]" on page 248
- "TAN function [Numeric]" on page 265

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the cotangent value of 0.52.

```
SELECT COT( 0.52 );
```

COUNT function [Aggregate]

Counts the number of rows in a group depending on the specified parameters.

Syntax 1

COUNT(
*
| *expression*
| **DISTINCT** *expression*
)

Syntax 2

COUNT(
{ * | *expression* }
) OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

***** Return the number of rows in each group.

expression The expression for which to return the number of rows.

DISTINCT expression The expression for which to return the number of distinct rows.

Remarks

Rows where the value is the NULL value are not included in the count.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "AVG function [Aggregate]" on page 107
- "SUM function [Aggregate]" on page 264

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns each unique city, and the number of rows with that city value.

```
SELECT City , COUNT(*) FROM Employees GROUP BY City;
```

COUNT_SET_BITS function [Bit array]

Returns a count of the number of bits set to 1 (TRUE) in the array.

Syntax

COUNT_SET_BITS(*bit-expression* **)**

Parameters

The bit array for which to determine the set bits.

Remarks

Returns NULL if *bit-expression* is NULL.

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following statement returns the value 4:

```
SELECT COUNT_SET_BITS( '00110011' );
```

The following statement returns the value 12:

```
SELECT COUNT_SET_BITS( '0011001111111111' );
```

COVAR_POP function [Aggregate]

Returns the population covariance of a set of number pairs.

Syntax 1

COVAR_POP(*dependent-expression*, *independent-expression* **)**

Syntax 2

COVAR_POP(*dependent-expression*, *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The following computation is then made:

```
(SUM( x * y ) - SUM( y ) * SUM( y ) / n ) / n
```

where *x* represents the *dependent-expression* and *y* represents the *independent-expression*.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "COVAR_SAMP function [Aggregate]" on page 132
- "SUM function [Aggregate]" on page 264

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example measures the strength of association between employees' age and salary. This function returns the value 73785.84005866687.

```
SELECT COVAR_POP( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

COVAR_SAMP function [Aggregate]

Returns the sample covariance of a set of number pairs.

Syntax 1

COVAR_SAMP(*dependent-expression*, *independent-expression* **)**

Syntax 2

COVAR_SAMP(*dependent-expression*, *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "COVAR_POP function [Aggregate]" on page 131
- "SUM function [Aggregate]" on page 264

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 74782.94600540561.

```
SELECT COVAR_SAMP( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees ;
```

CSCONVERT function [String]

Converts strings between character sets.

Syntax

CSCONVERT(
string-expression,
target-charset-string
[, *source-charset-string*] **)**

Parameters

string-expression The string.

target-charset-string The destination character set. *target-charset-string* can be one of the following:

- **os_charset** Alias for the character set used by the operating system hosting the database server.
- **char_charset** Alias for the CHAR character set used by the database.
- **nchar_charset** Alias for the NCHAR character set used by the database.
- **any other supported character set label** You can specify any of the SQL Anywhere supported character set labels.

source-charset The character set used by the original *string-expression*. The default is db_charset (the database character set). *source-charset-string* can be one of the following:

- **os_charset** Alias for the character set used by the operating system.
- **char_charset** Alias for the CHAR character set used by the database.
- **nchar_charset** Alias for the NCHAR character set used by the database.
- **any other supported character set label** You can specify any of the SQL Anywhere supported character set labels.

Remarks

You can view the list of character sets supported by SQL Anywhere by executing the following command at a command prompt:

```
dbinit -le
```

For more information about the character set labels you can use with this function, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

This fragment converts the mytext column from the Traditional Chinese character set to the Simplified Chinese character set:

```
SELECT CSCONVERT( mytext, 'cp936', 'cp950' )
FROM mytable;
```

This fragment converts the mytext column from the database character set to the Simplified Chinese character set:

```
SELECT CSCONVERT( mytext, 'cp936' )
FROM mytable;
```

If a file name is stored in the database, it is stored in the database's character set. If the server is going to read from or write to a file whose name is stored in a database (for example, in an external stored procedure), the file name must be explicitly converted to the operating system's character set before the file can be accessed. File names stored in the database and retrieved by the client are converted automatically to the client's character set, so explicit conversion is not necessary.

This fragment converts the value in the filename column from the database character set to the operating system character set:

```
SELECT CSCONVERT( filename, 'os_charset' )
FROM mytable;
```

A table contains a list of file names. An external stored procedure takes a file name from this table as a parameter and reads information directly out of that file. The following statement works when character set conversion is not required:

```
SELECT MYFUNC( filename )
FROM mytable;
```

where mytable is a table that contains a filename column. However, if you need to convert the file name to the character set of the operating system, you would use the following statement.

```
SELECT MYFUNC( csconvert( filename, 'os_charset' ) )
FROM mytable;
```


CUME_DIST function [Ranking]

Computes the relative position of one value among a group of rows. It returns a decimal value between 0 and 1.

Syntax

CUME_DIST( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Remarks

Composite sort keys are not currently allowed in the CUME_DIST function. You can use composite sort keys with any of the other rank functions.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "DENSE_RANK function [Ranking]" on page 151
- "PERCENT_RANK function [Ranking]" on page 213
- "RANK function [Ranking]" on page 221

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612

Example

The following example returns a result set that provides a cumulative distribution of the salaries of employees who live in California.

```
SELECT DepartmentID, Surname, Salary,
CUME_DIST() OVER (PARTITION BY DepartmentID
ORDER BY Salary DESC) "Rank"
FROM Employees
WHERE State IN ('CA');
```

Here is the result set:

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.000	0.333333333333333
200	Clark	45000.000	0.666666666666667
200	Overbey	39300.000	1

DATALENGTH function [System]

Returns the length, in bytes, of the underlying storage for the result of an expression.

Syntax

DATALENGTH(*expression* **)**

Parameters

expression *expression* is usually a column name. If *expression* is a string constant, you must enclose it in quotes.

Remarks

The return values of the DATALENGTH function are as follows:

Data type	DATALENGTH
SMALLINT	2
INTEGER	4
DOUBLE	8
CHAR	Length of the data
BINARY	Length of the data

This function supports NCHAR inputs and/or outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 27, the longest string in the CompanyName column.

```
SELECT MAX( DATALENGTH( CompanyName ) )
FROM Customers;
```

The following statement returns the value 22, the length of the string '8sdofinsv8s7a7s7gehe4h':

```
SELECT DATALENGTH( '8sdofinsv8s7a7s7gehe4h' );
```

DATE function [Date and time]

Converts the expression into a date, and removes any hours, minutes, or seconds.

For information about controlling the interpretation of date formats, see “date_order option [compatibility]” [*SQL Anywhere Server - Database Administration*].

Syntax

DATE(*expression* **)**

Parameters

expression The value to be converted to date format, typically a string.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 1999-01-02 as a date.

```
SELECT DATE( '1999-01-02 21:20:53' );
```

The following statement returns the create dates of all the objects listed in the SYSOBJECT system view:

```
SELECT DATE( creation_time ) FROM SYSOBJECT;
```

DATEADD function [Date and time]

Returns the date produced by adding a number of the date parts to a date.

Syntax

DATEADD(*date-part*, *numeric-expression*, *date-expression* **)**

date-part :
year | **quarter** | **month** | **week** | **day** | **dayofyear** | **hour** | **minute** | **second** | **millisecond**

Parameters

date-part The date part to be added to the date. For more information about date parts, see "Date parts" on page 95.

numeric-expression The number of date parts to be added to the date. The *numeric_expression* can be any numeric type, but the value is truncated to an integer.

date-expression The date to be modified.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value: 1995-11-02 00:00:00.000.

```
SELECT DATEADD( month, 102, '1987/05/02' );
```

DATEDIFF function [Date and time]

Returns the interval between two dates.

Syntax

DATEDIFF(*date-part*, *date-expression-1*, *date-expression-2* **)**

date-part :
year | **quarter** | **month** | **week** | **day** | **dayofyear** | **hour** | **minute** | **second** | **millisecond**

Parameters

date-part Specifies the date part in which the interval is to be measured. Choose one of the date objects listed above. For a complete list of date parts, see "Date parts" on page 95.

date-expression-1 The starting date for the interval. This value is subtracted from *date-expression-2* to return the number of *date-parts* between the two arguments.

date-expression-2 The ending date for the interval. *Date-expression-1* is subtracted from this value to return the number of *date-parts* between the two arguments.

Remarks

This function calculates the number of date parts between two specified dates. The result is a signed integer value equal to (date2 – date1), in date parts.

The DATEDIFF function results are truncated, not rounded, when the result is not an even multiple of the date part.

When you use **day** as the date part, the DATEDIFF function returns the number of midnights between the two times specified, including the second date but not the first.

When you use **month** as the date part, the DATEDIFF function returns the number of first-of-the-months between two dates, including the second date but not the first.

When you use **week** as the date part, the DATEDIFF function returns the number of Sundays between the two dates, including the second date but not the first.

For the smaller time units there are overflow values:

- **milliseconds** 24 days
- **seconds** 68 years
- **minutes** 4083 years
- **others** No overflow limit

The function returns an overflow error if you exceed these limits.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns 1.

```
SELECT DATEDIFF( hour, '4:00AM', '5:50AM' );
```

The following statement returns 102.

```
SELECT DATEDIFF( month, '1987/05/02', '1995/11/15' );
```

The following statement returns 0.


```
SELECT DATEDIFF( day, '00:00', '23:59' );
```

The following statement returns 4.

```
SELECT DATEDIFF( day,
   '1999/07/19 00:00',
   '1999/07/23 23:59' );
```

The following statement returns 0.

```
SELECT DATEDIFF( month, '1999/07/19', '1999/07/23' );
```

The following statement returns 1.

```
SELECT DATEDIFF( month, '1999/07/19', '1999/08/23' );
```

DATEFORMAT function [Date and time]

Returns a string representing a date expression in the specified format.

Syntax

DATEFORMAT(*datetime-expression*, *string-expression* **)**

Parameters

datetime-expression The datetime to be converted.

string-expression The format of the converted date.

For information about date format descriptions, see "timestamp_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

This function supports NCHAR inputs and/or outputs.

Remarks

Any allowable date format can be used for the string-expression.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Jan 01, 1989.

```
SELECT DATEFORMAT( '1989-01-01', 'Mmm dd, yyyy' );
```

DATENAME function [Date and time]

Returns the name of the specified part (such as the month June) of a datetime value, as a character string.

Syntax

DATENAME(*date-part*, *date-expression* **)**

Parameters

date-part The date part to be named. For a complete listing of allowed date parts, see "Date parts" on page 95.

date-expression The date for which the date part name is to be returned. The date must contain the requested *date-part*.

Remarks

The DATENAME function returns a string, even if the result is numeric, such as 23 for the day.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value May.

```
SELECT DATENAME( month, '1987/05/02' );
```

DATEPART function [Date and time]

Returns the value of part of a datetime value.

Syntax

DATEPART(*date-part*, *date-expression* **)**

Parameters

date-part The date part to be returned. For a complete listing of allowed date parts, see "Date parts" on page 95.

date-expression The date for which the part is to be returned.

Remarks

The date must contain the *date-part* field.

The numbers that correspond week days depend on the setting of the first_day_of_week option. By default Sunday=7.

See also

- "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*]
- "SET statement [T-SQL]" on page 658

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT DATEPART( month , '1987/05/02' );
```


DATETIME function [Date and time]

Converts an expression into a timestamp.

Syntax

DATETIME(*expression* **)**

Parameters

expression The expression to be converted. It is generally a string.

Remarks

Attempts to convert numerical values return an error.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a timestamp with value 1998-09-09 12:12:12.000.

```
SELECT DATETIME( '1998-09-09 12:12:12.000' );
```

DAY function [Date and time]

Returns an integer from 1 to 31.

Syntax

DAY(*date-expression* **)**

Parameters

date-expression The date.

Remarks

The integers 1 to 31 correspond to the day of the month in a date.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 12.

```
SELECT DAY( '2001-09-12' );
```

DAYNAME function [Date and time]

Returns the name of the day of the week from a date.

Syntax

DAYNAME(*date-expression* **)**

Parameters

date-expression The date.

Remarks

The English names are returned as: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Saturday.

```
SELECT DAYNAME ( '1987/05/02' );
```

DAYS function [Date and time]

A function that evaluates days. For specific details, see this function's usage.

Syntax 1: integer

DAYS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

DAYS(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of days to be added to the *datetime-expression.* If the *integer-expression* is negative, the appropriate number of days is subtracted from the timestamp. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a date or timestamp.

For information about casting data types, see "CAST function [Data type conversion]" on page 115.

Remarks

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of days since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two dates, this function returns the integer number of days between them. Instead, use the DATEDIFF function.

- If you give a date and an integer, this function adds the integer number of days to the specified date. Instead, use the DATEADD function.

This function ignores hours, minutes, and seconds.

See also

- "DATEDIFF function [Date and time]" on page 137
- "DATEADD function [Date and time]" on page 137

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the integer 729889.

```
SELECT DAYS( '1998-07-13 06:07:12' );
```

The following statements return the integer value –366, indicating that the second date is 366 days prior to the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT DAYS( '1998-07-13 06:07:12',
             '1997-07-12 10:07:12' );

SELECT DATEDIFF( day,
   '1998-07-13 06:07:12',
   '1997-07-12 10:07:12' );
```

The following statements return the timestamp 1999-07-14 00:00:00.000. It is recommended that you use the second example (DATEADD).

```
SELECT DAYS( CAST('1998-07-13' AS DATE ), 366 );

SELECT DATEADD( day, 366, '1998-07-13' );
```

DB_EXTENDED_PROPERTY function [System]

Returns the value of the given property. Allows an optional property-specific string parameter to be specified.

Syntax

DB_EXTENDED_PROPERTY(
{ *property-id* | *property-name* }
[**,** *property-specific-argument*
[**,** *database-id* | *database-name*]]
)

Parameters

property-id The database property ID to query.

property-name The database property name to query.

For a complete list of database properties, see "Database-level properties" [*SQL Anywhere Server - Database Administration*].

property-specific-argument The following database properties allow you to specify additional arguments, as noted below, to return specific information about the property.

- **CharSet property** Specify the name of a standard to obtain the default CHAR character set label for the standard. Possible values you can specify are: ASE, IANA, MIME, JAVA, WINDOWS, UTR22, IBM, and ICU. If no standard is specified, IANA is used as the default, unless the database connection was made through TDS, in which case ASE is the default.
- **CatalogCollation, Collation, and NcharCollation properties** When querying these properties, the following values can be specified as a *property-specific-argument* to return information specific to the collation:
  - **AccentSensitivity** Specify AccentSensitivity to obtain the accent sensitivity setting for the collation. For example, the following statement returns the accent sensitivity setting for the NCHAR collation:

    ```
    SELECT DB_EXTENDED_PROPERTY( 'NcharCollation', 'AccentSensitivity');
    ```

    Possible return values are: Ignore, Respect, and French. For a description of these values, see "Collation tailoring options" on page 376.
  - **CaseSensitivity** Specify CaseSensitivity to obtain the case sensitivity setting for the collation. Possible return values are: Ignore, Respect, UpperFirst, and LowerFirst. For a description of these values, see "Collation tailoring options" on page 376.
  - **PunctuationSensitivity** Specify PunctuationSensitivity to obtain the punctuation sensitivity setting for the collation. Possible return values are: Ignore, Primary, and Quaternary. For a description of these values, see "Collation tailoring options" on page 376.
  - **Properties** Specify Properties to obtain a string containing all of the tailoring options specified for the collation. For a description of the keywords and values in the returned string, see "Collation tailoring options" on page 376.
  - **Specification** Specify Specification to obtain a string containing the full collation specification used for the collation. For a description of the keywords and values in the returned string, see "Collation tailoring options" on page 376.
- **DriveType property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain its drive type. The value returned is one of the following: CD, FIXED, RAMDISK, REMOTE, REMOVABLE, or UNKNOWN. If nothing is specified, the drive type of the system dbspace is returned. If the specified dbspace doesn't exist, the property function returns NULL. If the name of a dbspace is specified and the ID of a database that isn't the database of the current connection is also specified, the function also returns NULL.
- **File property** Specify a dbspace name to obtain the file name of the database root file, including the path. If nothing is specified, information for the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.
- **FileSize property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain the size of the specified file. You can also specify temporary to return the size of the temporary dbspace, or translog to return the size of the log file. If nothing is specified, the size of the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.

- **FreePages property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain the number of free pages. You can also specify temporary to return the number of free pages in the temporary dbspace, or translog to return the number of free pages in the log file. If nothing is specified, the number of free pages in the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.
- **IOParallelism property** Specify a dbspace name to obtain the estimated number of simultaneous I/O operations supported by the dbspace. If a dbspace is not specified, the current system dbspace is used.
- **NextScheduleTime property** Specify an event name to obtain its next scheduled execution time.

database-id The database ID number, as returned by the DB_ID function. Typically, the database name is used.

database-name The name of the database, as returned by the DB_NAME function.

Remarks

Returns a value of type LONG VARCHAR. The current database is used if the second argument is omitted.

The DB_EXTENDED_PROPERTY function is similar to the DB_PROPERTY function except that it allows an optional *property-specific-argument* string parameter to be specified. The interpretation of *property-specific-argument* depends on the property ID or name specified in the first argument.

When comparing catalog strings such as table names and procedure names, the database server uses the CHAR collation. For the UCA collation, the catalog collation is the same as the CHAR collation but with the tailoring changed to be case-insensitive, accent-insensitive and with punctuation sorted in the primary level. For legacy collations, the catalog collation is the same as the CHAR collation but with the tailoring changed to be case-insensitive. While you cannot explicitly specify the tailoring used for the catalog collation, you can query the Specification property to obtain the full collation specification used by the database server for comparing catalog strings. Querying the Specification property can be useful if you need to exploit the difference between the CHAR and catalog collations. For example, suppose you have a punctuation-insensitive CHAR collation and you want to execute an upgrade script that defines a procedure called my_procedure, and that also attempts to delete an old version named myprocedure. The following statements cannot achieve the desired results because my_procedure is equivalent to myprocedure, using the CHAR collation:

```
CREATE PROCEDURE my_procedure() ... ;
IF EXISTS ( SELECT * FROM SYS.SYSPROCEDURE WHERE proc_name = 'myprocedure' )
THEN DROP PROCEDURE myprocedure
END IF;
```

Instead, you could execute the following statements to achieve the desired results:

```
CREATE PROCEDURE my_procedure() ... ;
IF EXISTS ( SELECT * FROM SYS.SYSPROCEDURE
   WHERE COMPARE( proc_name, 'myprocedure', DB_EXTENDED_PROPERTY
( 'CatalogCollation', 'Specification' ) ) = 0 )
THEN DROP PROCEDURE myprocedure
END IF;
```

See also

- "DB_ID function [System]" on page 146
- "DB_NAME function [System]" on page 147

- "Database-level properties" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_PROPERTY function [System]" on page 122
- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the file size of the system dbspace, in pages.

```
SELECT DB_EXTENDED_PROPERTY( 'FileSize' );
```

The following statement returns the file size of the transaction log, in pages.

```
SELECT DB_EXTENDED_PROPERTY( 'FileSize', 'translog' );
```

The following statement returns the case sensitivity setting for the NCHAR collation:

```
SELECT DB_EXTENDED_PROPERTY( 'NcharCollation',' CaseSensitivity' );
```

The statement `SELECT DB_EXTENDED_PROPERTY ( 'Collation', 'Properties' );` returns the tailoring options specified for the database CHAR collation:

```
'CaseSensitivity=Ignore'
```

The statement `SELECT DB_EXTENDED_PROPERTY( 'NCharCollation', 'Specification' );` returns the full collation specification for the database NCHAR collation:

```
'UCA
(CaseSensitivity=Ignore;AccentSensitivity=Ignore;PunctuationSensitivity=Prima
ry)'
```

DB_ID function [System]

Returns the database ID number.

Syntax

DB_ID([*database-name*] **)**

Parameters

database-name A string containing the database name. If no *database-name* is supplied, the ID number of the current database is returned.

See also

- "global_database_id option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the value 0, when executed against the SQL Anywhere sample database as the sole database on the server.

```
SELECT DB_ID( 'demo' );
```

The following statement returns the value 0 if executed against the only running database.

```
SELECT DB_ID();
```

DB_NAME function [System]

Returns the name of a database with a given ID number.

Syntax

DB_NAME([*database-id*] **)**

Parameters

database-id The ID of the database. The *database-id* must be a numeric expression.

Remarks

If no database ID is supplied, the name of the current database is returned.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the database name demo, when executed against the SQL Anywhere sample database as the sole database on the server.

```
SELECT DB_NAME( 0 );
```

DB_PROPERTY function [System]

Returns the value of the given property.

Syntax

DB_PROPERTY(
{ *property-id* | *property-name* }
[**,** *database-id* | *database-name*]
)

Parameters

property-id The database property ID.

property-name The database property name.

database-id The database ID number, as returned by the DB_ID function. Typically, the database name is used.

database-name The name of the database, as returned by the DB_NAME function.

Remarks

Returns a string. The current database is used if the second argument is omitted.

See also

- "DB_ID function [System]" on page 146
- "DB_NAME function [System]" on page 147
- "Database-level properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the page size of the current database, in bytes.

```
SELECT DB_PROPERTY( 'PAGESIZE' );
```

DECOMPRESS function [String]

Decompresses the string and returns a LONG BINARY value.

Syntax

DECOMPRESS(*string-expression* [**,** *compression-algorithm-alias*] **)**

Parameters

string-expression The string to decompress. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

compression-algorithm-alias Alias (string) for the algorithm to use for decompression. The supported values are zip and gzip (both are based on the same algorithm, but use different headers and trailers).

Zip is a widely supported compression algorithm. Gzip is compatible with the gzip utility on Unix, whereas the zip algorithm is not.

If no algorithm is specified, the function attempts to detect which algorithm was used to compress the string. If the incorrect algorithm is specified, or the correct algorithm cannot be detected, the string is not decompressed.

For more information about compression, see "COMPRESS function [String]" on page 121.

Remarks

The DECOMPRESS function returns a LONG BINARY value. This function can be used to decompress a value that was compressed using the COMPRESS function.

You do not need to use the DECOMPRESS function on values that are stored in a compressed column. Compression and decompression of values in a compressed column are handled automatically by the database server. See "Choosing whether to compress columns" [*SQL Anywhere Server - SQL Usage*].

See also

- "COMPRESS function [String]" on page 121
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example uses the DECOMPRESS function to decompress values from the Attachment column of a fictitious table, TableA:

```
SELECT DECOMPRESS ( Attachment, 'gzip' )
FROM TableA;
```

Since DECOMPRESS returns binary values, if the original values were of a character type, such as LONG VARCHAR, a CAST can be applied to return human-readable values:

```
SELECT CAST ( DECOMPRESS ( Attachment, 'gzip' )
AS LONG VARCHAR ) FROM TableA;
```

DECRYPT function [String]

Decrypts the string using the supplied key and returns a LONG BINARY value.

Syntax

DECRYPT(*string-expression*, *key*
[, *algorithm*]
)

Parameters

string-expression The string to be decrypted. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

key The encryption key (string) required to decrypt the *string-expression*. This must be the same encryption key that was used to encrypt the *string-expression* to obtain the original value that was encrypted. This parameter is case sensitive, even in case-insensitive databases.

> **Caution**
> Protect your key. Be sure to store a copy of your key in a safe location. A lost key will result in the encrypted data becoming completely inaccessible, from which there is no recovery.

algorithm This optional parameter specifies the algorithm used to decrypt the *string-expression*. The *string-expression* must be decrypted using the same algorithm with which it was encrypted. The algorithm used to implement SQL Anywhere strong encryption is Rijndael: a block encryption algorithm chosen as

the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

On any platform that supports FIPS, you can also specify a separate FIPS-approved AES algorithm for strong encryption using the AES_FIPS type. When the database server is started with the -fips option, you can run databases encrypted with AES or AES_FIPS strong encryption, but not databases encrypted with simple encryption. Unencrypted databases can also be started on the server when -fips is specified.

Remarks

You can use the DECRYPT function to decrypt a *string-expression* that was encrypted with the ENCRYPT function. This function returns a LONG BINARY value with the same number of bytes as the input string.

To successfully decrypt a *string-expression*, you must use the same encryption key that was used to encrypt the data. If you specify an incorrect encryption key, an error is generated. A lost key will result in inaccessible data, from which there is no recovery.

See also

- "ENCRYPT function [String]" on page 154
- "Encrypting portions of a database" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example decrypts a user's password from the user_info table. The CAST function is used to convert the password back to a CHAR data type because the DECRYPT function converts values to the LONG BINARY data type.

```
SELECT CAST( DECRYPT( user_pwd, '8U3dkA' ) AS CHAR(100) ) FROM user_info;
```

DEGREES function [Numeric]

Converts a number from radians to degrees.

Syntax

DEGREES(*numeric-expression* **)**

Parameters

numeric-expression An angle in radians.

Remarks

The DEGREES function returns the degrees of the angle given by *numeric-expression*.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 29.79380534680281.

```
SELECT DEGREES( 0.52 );
```

DENSE_RANK function [Ranking]

Calculates the rank of a value in a partition. In the case of tied values, the DENSE_RANK function does not leave gaps in the ranking sequence.

Syntax

DENSE_RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "CUME_DIST function [Ranking]" on page 135
- "PERCENT_RANK function [Ranking]" on page 213
- "RANK function [Ranking]" on page 221

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612

Example

The following example returns a result set that provides a ranking of the employees' salaries in Utah and New York. Although 19 records are returned in the result set, only 18 rankings are listed because of a 7th-place tie between the 7th and 8th employee in the list, who have identical salaries. Instead of ranking the 9th employee as '9', the employee is listed as '8' because the DENSE_RANK function does not leave gaps in the ranks.

```
SELECT DepartmentID, Surname, Salary, State,
DENSE_RANK() OVER (ORDER BY Salary DESC) AS SalaryRank
FROM Employees
WHERE State IN ('NY','UT');
```

Here is the result set:

Surname	Salary	State	SalaryRank
Shishov	72995.000	UT	1
Wang	68400.000	UT	2
Cobb	62000.000	UT	3
Morris	61300.000	UT	4
Davidson	57090.000	NY	5
Martel	55700.000	NY	6
Blaikie	54900.000	NY	7
Diaz	54900.000	UT	7
Driscoll	48023.000	UT	8
Hildebrand	45829.000	UT	9
Whitney	45700.000	NY	10
Guevara	42998.000	NY	11
Soo	39075.000	NY	12
Goggin	37900.000	UT	13
Wetherby	35745.000	NY	14
Ahmed	34992.000	NY	15
Rebeiro	34576.000	UT	16
Bigelow	31200.000	UT	17
Lynch	24903.000	UT	18

DIFFERENCE function [String]

Returns the difference in the SOUNDEX values between the two string expressions.

Syntax

DIFFERENCE (*string-expression-1*, *string-expression-2* **)**

Parameters

string-expression-1 The first SOUNDEX argument.

string-expression-2 The second SOUNDEX argument.

Remarks

The DIFFERENCE function compares the SOUNDEX values of two strings and evaluates the similarity between them, returning a value from 0 through 4, where 4 is the best match.

This function always returns some value. The result is NULL only if one of the arguments are NULL.

See also

- "SOUNDEX function [String]" on page 253
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 3.

```
SELECT DIFFERENCE( 'test', 'chest' );
```

DOW function [Date and time]

Returns a number from 1 to 7 representing the day of the week of a date, where Sunday=1, Monday=2, and so on.

Syntax

DOW(*date-expression* **)**

Parameters

date-expression The date to evaluate.

Remarks

The DOW function is not affected by the value specified for the first_day_of_week database option. For example, even if first_day_of_week is set to Monday, the DOW function returns a 2 for Monday.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT DOW( '1998-07-09' );
```

The following statement queries the Employees table and returns the employees StartDate, expressed as the number of the day of the week:

```
SELECT DOW( StartDate ) FROM Employees;
```


ENCRYPT function [String]

Encrypts the specified values using the supplied encryption key and returns a LONG BINARY value.

Syntax

ENCRYPT(*string-expression*, *key*
[, *algorithm*]
)

Parameters

string-expression The data to be encrypted. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.

key The encryption key used to encrypt the *string-expression*. This same key must be used to decrypt the value to obtain the original value. This parameter is case sensitive, even in case-insensitive databases.

As with most passwords, it is best to choose a key value that cannot be easily guessed. It is recommended that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase, and includes numbers, letters and special characters. You will require this key each time you want to decrypt the data.

> **Caution**
> Protect your key. Be sure to store a copy of your key in a safe location. A lost key will result in the encrypted data becoming completely inaccessible, from which there is no recovery.

algorithm This optional parameter specifies the algorithm used to encrypt the *string-expression*. The *string-expression* must be encrypted using the same algorithm with which it will be decrypted. The algorithm used to implement SQL Anywhere strong encryption is Rijndael: a block encryption algorithm chosen as the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

On any platform that supports FIPS,, you can also specify a separate FIPS-approved AES algorithm for strong encryption using the AES_FIPS algorithm.

Remarks

This function returns a LONG BINARY value, which is at most 31 bytes longer than the input *string-expression*. The value returned by this function is not human-readable. You can use the DECRYPT function to decrypt a *string-expression* that was encrypted with the ENCRYPT function. To successfully decrypt a *string-expression*, you must use the same encryption key and algorithm that were used to encrypt the data. If you specify an incorrect encryption key, an error is generated. A lost key will result in inaccessible data, from which there is no recovery.

If you are storing encrypted values in a table, the column should be BINARY or LONG BINARY so that character set conversion is not performed on the data.

See also

- "DECRYPT function [String]" on page 149
- "Encrypting portions of a database" [*SQL Anywhere Server - Database Administration*]

- "-fips server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** SQL foundation feature outside of core SQL.

Example

The following trigger encrypts the user_pwd column of the user_info table. This column contains users' passwords, and the trigger fires whenever the password value is changed.

```
CREATE TRIGGER encrypt_updated_pwd
BEFORE UPDATE OF user_pwd
ON user_info
REFERENCING NEW AS new_pwd
FOR EACH ROW
BEGIN
    SET new_pwd.user_pwd=ENCRYPT( new_pwd.user_pwd, '8U3dkA' );
END;
```

ERRORMSG function [Miscellaneous]

Provides the error message for the current error, or for a specified SQLSTATE or SQLCODE value.

Syntax

ERRORMSG([*sqlstate* | *sqlcode*] **)**

sqlstate: *string*

sqlcode: *integer*

Parameters

sqlstate The SQLSTATE value for which the error message is to be returned.

sqlcode The SQLCODE value for which the error message is to be returned.

Return value

A string containing the error message. If no argument is supplied, the error message for the current state is supplied. Any substitutions (such as table names and column names) are made.

If an argument is supplied, the error message for the supplied SQLSTATE or SQLCODE is returned, with no substitutions. Table names and column names are supplied as placeholders (%1).

See also

- "Error messages sorted by SQLSTATE" [*SQL Anywhere 10 - Error Messages*]
- "Error messages sorted by SQL Anywhere SQLCODE" [*SQL Anywhere 10 - Error Messages*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the error message for SQLCODE -813.

```
SELECT ERRORMSG( -813 );
```

ESTIMATE function [Miscellaneous]

Provides selectivity estimates for the query optimizer, based on specified parameters.

Syntax

ESTIMATE(*column-name* [, *value* [, *relation-string*]] **)**

Parameters

column-name The column used in the estimate.

value The value to which the column is compared. The default is NULL.

relation-string The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: =, >, <, >= , <=, <>, !=, !<, and !>. The default is =.

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- "INDEX_ESTIMATE function [Miscellaneous]" on page 183
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 156

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the percentage of EmployeeID values estimated to be greater than 200. The precise value depends on the actions you have carried out on the database.

```
SELECT FIRST ESTIMATE( EmployeeID, 200, '>' )
   FROM Employees;
```

ESTIMATE_SOURCE function [Miscellaneous]

Provides the source for selectivity estimates used by the query optimizer.

Syntax

ESTIMATE_SOURCE(
column-name
[, *value*
[, *relation-string*]]
)

Parameters

column-name The name of the column that is being investigated.

value The value to which the column is compared. The default is NULL.

relation-string The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: =, >, <, >= , <=, <>, !=, !<, and !>. The default is =.

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

Return value

The source of the selectivity estimate can be one of the following:

- **Statistics** is used as the source when you have specified a value, and there is a stored statistic available that estimates the average selectivity of the value in the column. The statistic is available only when the selectivity of the value is a significant enough number that it is stored in the statistics. Currently, a value is deemed significant if it occurs in at least 1% of the rows.
- **Column** is similar to Statistics, except that the selectivity of the value occurs in less than 1% of the rows. In this case, the selectivity that is used is the average of all values that have been stored in the statistics that occur in less than 1% of rows.
- **Guess** is returned when there is no relevant index to use, and no statistics have been collected for the column. In this case, built-in guesses are used.
- **Column-column** is returned when the estimate that is used is the selectivity of a join. In this case, the estimate is calculated as the number of rows in the joined result set divided by the number of rows in the Cartesian product of the two tables.
- **Index** is used as the source when there are no statistics available to estimate the selectivity, but there is an index which can be probed to estimate selectivity.
- **User** is returned when there is a user supplied estimate, and the user_estimates database option is not set to Disabled.

  For more information, see "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*].
- **Computed** is returned when statistics are computed by the optimizer based on other information. For example, SQL Anywhere does not maintain statistics on multiple columns, so if you want an estimate on a multiple column equation, such as x=5 and y=10, and there are statistics on the columns x and y, then the optimizer creates an estimate by multiplying the estimated selectivity for each column.
- **Always** is used when the test is by definition true. For example, if the value is 1=1.
- **Combined** is used when the optimizer uses more than one of the above sources, and combines them.
- **Bounded** can qualify one of the other sources. This indicates that SQL Anywhere has placed an upper and/or lower bound on the estimate. The optimizer does this to keep estimates within logical bounds.

For example, it ensures that an estimate is not greater than 100%, or that the selectivity is not less than one row.

See also

- "ESTIMATE function [Miscellaneous]" on page 156
- "INDEX_ESTIMATE function [Miscellaneous]" on page 183

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Index, which means that the query optimizer probed an index to estimate the selectivity.

```
SELECT FIRST ESTIMATE_SOURCE( EmployeeID, 200, '>' )
FROM Employees;
```

EVENT_CONDITION function [System]

Specifies when an event handler is triggered.

Syntax

EVENT_CONDITION(*condition-name* **)**

Parameters

condition-name The condition triggering the event. The possible values are preset in the database, and are case insensitive. Each condition is valid only for certain event types. The conditions and the events for which they are valid are as follows:

Condition name	Units	Valid for…	Comments
DBFreePercent	n/a	DBDiskSpace	
DBFreeSpace	MB	DBDiskSpace	
DBSize	MB	GrowDB	
ErrorNumber	n/a	RAISERROR	
IdleTime	seconds	ServerIdle	
Interval	seconds	All	Time since handler last executed
LogFreePercent	n/a	LogDiskSpace	
LogFreeSpace	MB	LogDiskSpace	
LogSize	MB	GrowLog	

Condition name	Units	Valid for…	Comments
RemainingValues	integer	GlobalAutoincrement	The number of remaining values
TempFreePercent	n/a	TempDiskSpace	
TempFreeSpace	MB	TempDiskSpace	
TempSize	MB	GrowTemp	

Remarks

The EVENT_CONDITION function returns NULL when not called from an event.

See also

- "CREATE EVENT statement" on page 390

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following event definition uses the EVENT_CONDITION function:

```
CREATE EVENT LogNotifier
TYPE LogDiskSpace
WHERE event_condition( 'LogFreePercent' ) < 50
HANDLER
BEGIN
   MESSAGE 'LogNotifier message'
END;
```

EVENT_CONDITION_NAME function [System]

Can be used to list the possible parameters for EVENT_CONDITION.

Syntax

EVENT_CONDITION_NAME(*integer* **)**

Parameters

integer Must be greater than or equal to zero.

Remarks

You can use the EVENT_CONDITION_NAME function to obtain a list of all arguments for the EVENT_CONDITION function by looping over integers until the function returns NULL.

The EVENT_CONDITION_NAME function returns NULL when not called from an event.

See also

- "CREATE EVENT statement" on page 390

Standards and compatibility

- **SQL/2003** Vendor extension.

EVENT_PARAMETER function [System]

Provides context information for event handlers.

Syntax

EVENT_PARAMETER(*context-name* **)**

context-name:
'AppInfo'
| **'ConnectionID'**
| **DisconnectReason**
| **'EventName'**
| **'Executions'**
| **'MirrorServerName'**
| **'NumActive'**
| **'ScheduleName'**
| **'TableName'**
| **'User'**
| *condition-name*

Parameters

context-name One of the preset strings. The strings are case insensitive, and carry the following information:

- **AppInfo** The value of the AppInfo connection property for the connection that caused the event to be triggered. You can see the value of the property from outside of the context of the event by using the following statement:

  ```
  SELECT connection_property( 'AppInfo' );
  ```

  This parameter is valid for Connect, Disconnect, ConnectFailed, BackupEnd, and RAISERROR events. The AppInfo string contains the machine name and application name of the client connection for embedded SQL, ODBC, OLE DB, ADO.NET, and iAnywhere JDBC driver connections.

- **ConnectionId** The connection ID of the connection that caused the event to be triggered.

- **DisconnectReason** A string indicating the reason the connect was terminated. This parameter is valid only for Disconnect events. Possible results include:

  - **from client** The client application disconnected.
  - **drop connection** A DROP CONNECTION statement was executed.
  - **liveness** No liveness packets were received for the period specified by the -tl server option.
  - **inactive** No requests were received for the period specified by the -ti server option.
  - **connect failed** A connection attempt failed.

- **EventName** The name of the event that has been triggered.
- **Executions** The number of times the event handler has been executed.
- **MirrorServerName** The name of the mirror or arbiter server that lost its connection to the primary server in a database mirroring system.
- **NumActive** The number of active instances of an event handler. This is useful if you want to limit an event handler so that only one instance executes at any given time.
- **ScheduleName** The name of the schedule which caused an event to be fired. If the event was fired manually using TRIGGER EVENT or as a system event, the result will be an empty string. If the schedule was not assigned a name explicitly when it was created, its name will be the name of the event.
- **TableName** The name of the table, for use with RemainingValues.
- **User** The user ID for the user that caused the event to be triggered.

In addition, you can access any of the valid *condition-name* arguments to the EVENT_CONDITION function from the EVENT_PARAMETER function.

The following table indicates which context-name values are valid for which system event types.

Context-name value	Valid system event types
AppInfo	BackupEnd, "Connect", ConnectFailed, "Disconnect", "RAISERROR", user events
ConnectionID	BackupEnd, "Connect", "Disconnect", Global Autoincrement, "RAISERROR", user events
DisconnectReason	"Disconnect"
EventName	all
Executions	all
NumActive	all
TableName	GlobalAutoincrement
User	BackupEnd, "Connect", ConnectFailed, "Disconnect", GlobalAutoincrement, "RAISERROR", user events

See also

- “EVENT_CONDITION function [System]” on page 158
- “CREATE EVENT statement” on page 390
- “TRIGGER EVENT statement” on page 692

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example shows how to pass a string parameter to an event. The event displays the time it was triggered on the server console.

```
CREATE EVENT ev_PassedParameter
HANDLER
BEGIN
  MESSAGE 'ev_PassedParameter - was triggered at ' || event_parameter
( 'time' );
END;
TRIGGER EVENT ev_PassedParameter( "Time"=string(current timestamp ) );
```

EXP function [Numeric]

Returns the exponential function, e to the power of a number.

Syntax

EXP(*numeric-expression* **)**

Parameters

numeric-expression The exponent.

Remarks

The EXP function returns the exponential of the value specified by *numeric-expression*.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the value 3269017.3724721107.

```
SELECT EXP( 15 );
```

EXPERIENCE_ESTIMATE function [Miscellaneous]

This function is the same as the ESTIMATE function, except that it always looks in the frequency table.

Syntax

EXPERIENCE_ESTIMATE(
column-name
[**,** *value*
[**,** *relation-string*]]
)

Parameters

column-name The name of the column that is being investigated.

value The value to which the column is compared.

relation-string The comparison operator used for the comparison. Possible values for this parameter are: =, >, <, >= , <=, <>, !=, !<, and !>. The default is =.

Remarks

If *value* is NULL then the relation strings = and != are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- "ESTIMATE function [Miscellaneous]" on page 156

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns NULL.

```
SELECT DISTINCT EXPERIENCE_ESTIMATE( EmployeeID, 200, '>' )
FROM Employees;
```

EXPLANATION function [Miscellaneous]

Returns the plan optimization strategy of a SQL statement.

Syntax

EXPLANATION(
string-expression
[*cursor-type*],
[*update-status*]
)

Parameters

string-expression The SQL statement, which is commonly a SELECT statement, but can also be an UPDATE or DELETE statement.

cursor-type A cursor type, expressed as a string. Possible values are asensitive, insensitive, sensitive, or keyset-driven. If *cursor-type* is not specified, asensitive is used by default.

update-status A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is the same as READ-WRITE.

Remarks

The optimization is returned as a string.

This information can help you decide which indexes to add or how to structure your database for better performance.

In Interactive SQL, you can view the plan for any SQL statement on the Plan tab in the Results pane.

See also

- "Query access plans in UltraLite" [*UltraLite - Database Management and Reference*]
- "Reading execution plans" [*SQL Anywhere Server - SQL Usage*]
- "PLAN function [Miscellaneous]" on page 214
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 169

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT EXPLANATION( 'SELECT * FROM Departments WHERE DepartmentID > 100' );
```

The following statement returns a string containing the short form of the textual plan for an INSENSITIVE cursor over the query 'select * from Departments where ….'.

```
SELECT EXPLANATION( 'SELECT * FROM Departments WHERE DepartmentID > 100',
   'insensitive', 'read-only' );
```

EXPRTYPE function [Miscellaneous]

Returns a string that identifies the data type of an expression.

Syntax

EXPRTYPE(*string-expression*, *integer-expression* **)**

Parameters

string-expression A SELECT statement. The expression whose data type is to be queried must appear in the select list. If the string is not a valid SELECT statement, NULL is returned.

integer-expression The position in the select list of the desired expression. The first item in the select list is numbered 1. If the integer-expression value does not correspond to a SELECT list item, NULL is returned.

See also

- "SQL Data Types" on page 47
- "sa_describe_query system procedure" on page 860

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns `smallint` when executed against the SQL Anywhere sample database.

```
SELECT EXPRTYPE( 'SELECT LineID FROM SalesOrderItems', 1 );
```

FIRST_VALUE function [Aggregate]

Returns values from the first row of a window.

Syntax

FIRST_VALUE(*expression* [**IGNORE NULLS**] **)**
OVER (*window-spec* **)**

window-spec : see the Remarks section below

Parameters

expression The expression to evaluate. For example, a column name.

Remarks

The FIRST_VALUE function allows you to select the first value (according to some ordering) in a table, without having to use a self-join. This is valuable when you want to use the first value as the baseline in calculations.

The FIRST_VALUE function takes the first record from the window. Then, the *expression* is computed against the first record and results are returned.

If IGNORE NULLS is specified, the first non-NULL value of *expression* is returned. If IGNORE NULLs is not specified, the first value is returned whether or not it is NULL.

The FIRST_VALUE function is different from most other aggregate functions in that it can only be used with a window specification.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Window aggregate functions" [*SQL Anywhere Server - SQL Usage*]
- "LAST_VALUE function [Aggregate]" on page 187

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the relationship, as a percentage, between each employee's salary and that of the most recently hired employee in the same department:

```
SELECT DepartmentID, EmployeeID,
       100 * Salary / ( FIRST_VALUE( Salary ) OVER (
                           PARTITION BY DepartmentID  ORDER BY StartDate
DESC ) )
            AS percentage
    FROM Employees;
```

In the result set below, since employee 1658 is the first row for department 500, you know that they are the most recent hire in that department; therefore, their percentage is set to 100%. Then, percentages for the remaining employees in department 500 are calculated relative to that of employee 1658. For example, employee 1570 earns approximately 139% of what employee 1658 earns.

If another employee in the same department makes the same salary as the most recent hire, they will have a percentage of 100 as well.

DepartmentID	EmployeeID	percentage
500	1658	100
500	1615	110.4284624
500	1570	138.8427097
500	1013	109.5851905
500	921	167.4497049
500	868	113.2393688
500	750	137.7344095
500	703	222.8679276
500	191	119.6642975
400	1751	100
400	1740	99.705647
400	1684	130.969936
400	1643	83.9734797
400	1607	175.1828989
400	1576	197.0164609
...	...	...

FLOOR function [Numeric]

Returns the floor of (largest integer not greater than) a number.

Syntax

FLOOR(*numeric-expression* **)**

Parameters

numeric-expression The value, usually a FLOAT.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "CEILING function [Numeric]" on page 115

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements returns a Floor value of 123

```
SELECT FLOOR (123);
```

The following statements returns a Floor value of 123

```
SELECT FLOOR (123.45);
```

The following statements returns a Floor value of -124

```
SELECT FLOOR (-123.45);
```

GET_BIT function [Bit array]

Returns the value (1 or 0) of a specified bit in a bit array.

Syntax

GET_BIT(*bit-expression*, *position* **)**

Parameters

bit-expression The bit array containing the bit.

position The position of the bit for which to return the status.

Remarks

The first bit in the array is considered position 1.

If *position* exceeds the length of the array, 0 (false) is returned.

See also

- "Bitwise operators" on page 13
- "SET_BIT function [Bit array]" on page 244
- "SET_BITS function [Aggregate]" on page 245
- "sa_get_bits system procedure" on page 869

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following statement returns the value 1:

```
SELECT GET_BIT( '00110011' , 4 );
```

The following statement returns the value 0:

```
SELECT GET_BIT( '00110011' , 5 );
```

GET_IDENTITY function [Miscellaneous]

Allocates values to an autoincrement column. This is an alternative to using autoincrement to generate numbers.

Syntax

GET_IDENTITY(*table_name* [**,** *number_to_allocate*] **)**

Parameters

table_name A string indicating the name of the table, including, optionally, the owner name.

number_to_allocate The starting number to allocate for the identity. Default is 1.

Remarks

Using autoincrement or global autoincrement is still the most efficient way to generate IDs, but this function is provided as an alternative. The function assumes that the table has an autoincrement column defined. It returns the next available value that would be generated for the table's autoincrement column, and reserves that value so that no other connection will use it by default.

The function returns an error if the table is not found, and returns NULL if the table has no autoincrement column. If there is more than one autoincrement column, it uses the first one it finds.

number_to_allocate is the number of values to reserve. If *number_to_allocate* is greater than 1, the function also reserves the remaining values. The next allocation uses the current number plus the value of *number_to_allocate*. This allows the application to execute the GET_IDENTITY function less frequently.

No COMMIT is required after executing the GET_IDENTITY function, and so it can be called using the same connection that is used to insert rows. If ID values are required for several tables, they can be obtained using a single SELECT that includes multiple calls to the GET_IDENTITY function, as in the example.

The GET_IDENTITY function is non-deterministic function; successive calls to it may return different values. The optimizer does not cache the results of the GET_IDENTITY function.

For more information about non-deterministic functions, see "Function caching" [*SQL Anywhere Server - SQL Usage*].

See also

- "CREATE TABLE statement" on page 450
- "ALTER TABLE statement" on page 332
- "NUMBER function [Miscellaneous]" on page 211

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the next available value for the table's autoincrement column, and reserves that number as well as the following nine values:

```
SELECT GET_IDENTITY( 'GROUPO.T2', 10 );
```

GETDATE function [Date and time]

Returns the current year, month, day, hour, minute, second and fraction of a second.

Syntax

GETDATE( )

Remarks

The accuracy is limited by the accuracy of the system clock.

The information the GETDATE function returns is equivalent to the information returned by the NOW function and the CURRENT TIMESTAMP special value.

See also

- "NOW function [Date and time]" on page 210
- "CURRENT TIMESTAMP special value" on page 31

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the system date and time.

```
SELECT GETDATE( );
```

GRAPHICAL_PLAN function [Miscellaneous]

Returns the plan optimization strategy of a SQL statement in XML format, as a string.

Syntax

GRAPHICAL_PLAN(
string-expression
[**,** *statistics-level*
[**,** *cursor-type*
[**,** *update-status*]]] **)**

Parameters

string-expression The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE statement.

statistics-level An integer. *Statistics-level* can be one of the following values:

Value	Description
0	Optimizer estimates only (default).
2	Detailed statistics including node statistics.
3	Detailed statistics.

cursor-type A cursor type, expressed as a string. Possible values are: asensitive, insensitive, sensitive, or keyset-driven. If *cursor-type* is not specified, asensitive is used by default.

update-status A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is exactly the same as READ-WRITE.

See also

- "PLAN function [Miscellaneous]" on page 214
- "EXPLANATION function [Miscellaneous]" on page 163

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following Interactive SQL example passes a SELECT statement as a string parameter and returns the plan for executing the query. It saves the plan in the file *plan.xml*.

```
SELECT GRAPHICAL_PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100' );
OUTPUT TO plan.xml FORMAT FIXED;
```


The following statement returns a string containing the graphical plan for a keyset-driven, updatable cursor over the query `SELECT * FROM Departments WHERE DepartmentID > 100`. It also causes the server to annotate the plan with actual execution statistics, in addition to the estimated statistics that were used by the optimizer.

```
SELECT GRAPHICAL_PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100',
   2,
   'keyset-driven', 'for update' );
```

In Interactive SQL, you can view the plan for any SQL statement on the Plan tab in the Results pane.

GREATER function [Miscellaneous]

Returns the greater of two parameter values.

Syntax

GREATER(*expression-1*, *expression-2* **)**

Parameters

expression-1 The first parameter value to be compared.

expression-2 The second parameter value to be compared.

Remarks

If the parameters are equal, the first is returned.

See also

- "LESSER function [Miscellaneous]" on page 191

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 10.

```
SELECT GREATER( 10, 5 ) FROM dummy;
```

GROUPING function [Aggregate]

Identifies whether a column in a GROUP BY operation result set is NULL because it is part of a subtotal row, or NULL because of the underlying data.

Syntax

GROUPING(*group-by-expression* **)**

Parameters

group-by-expression An expression appearing as a grouping column in the result set of a query that uses a GROUP BY clause. This function can be used to identify subtotal rows added to the result set by a ROLLUP or CUBE operation.

Return value

- **1** Indicates that *group-by-expression* is NULL because it is part of a subtotal row. The column is not a prefix column for that row.
- **0** Indicates that *group-by-expression* is a prefix column of a subtotal row.

See also

- "Using ROLLUP" [*SQL Anywhere Server - SQL Usage*]
- "Using CUBE" [*SQL Anywhere Server - SQL Usage*]
- "GROUP BY GROUPING SETS" [*SQL Anywhere Server - SQL Usage*]
- "SELECT statement" on page 648
- "Detecting placeholder NULLs using the GROUPING function" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL foundation feature (T611) outside of core SQL.

Example

For examples of this function in use, see "Detecting placeholder NULLs using the GROUPING function" [*SQL Anywhere Server - SQL Usage*].

HASH function [String]

Returns the specified value in hashed form.

Syntax

HASH(*string-expression*[**,** *algorithm*] **)**

Parameters

string-expression The value to be hashed. This parameter is case sensitive, even in case-insensitive databases.

algorithm The algorithm to use for the hash. Possible values include: MD5, SHA1, SHA1_FIPS, SHA256, SHA256_FIPS. By default, the MD5 algorithm is used.

> **Note**
> The FIPS algorithms are only for use on systems using FIPS 140-2 certified software from Certicom. You must specify the -fips option when starting the database server to use the FIPS algorithms.

Remarks

Using a hash converts the value to a byte sequence that is unique to each value passed to the function.

If the server was started with the -fips option, the algorithm used, or the behavior, may be different, as follows:

- SHA1_FIPS is used if SHA1 is specified
- SHA256_FIPS is used if SHA256 is specified
- an error is returned if MD5 is specified

Following are the return types, depending on the algorithm used:

- MD5 returns a VARCHAR(32)
- SHA1 returns a VARCHAR(40)
- SHA1_FIPS returns a VARCHAR(40)
- SHA256 returns a VARCHAR(40)
- SHA256_FIPS returns a VARCHAR(40)

Caution
All of the algorithms are one-way hashes. It is not possible to re-create the original string from the hash.

See also

- "String functions" on page 99
- "-fips server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a table called user_info to store information about the users of an application, including their user ID and password. One row is also inserted into the table. The password is hashed using the HASH function and the SHA256 algorithm. Storing hashed passwords in this way can be useful if you do not want to store passwords in clear text, yet you have an external application that needs to compare passwords.

```
CREATE TABLE user_info (
  employee_id   INTEGER NOT NULL PRIMARY KEY,
  user_name CHAR(80),
  user_pwd CHAR(80) );
INSERT INTO user_info
  VALUES ( '1', 's_phillips', HASH( 'mypass', 'SHA256' ) );
```

HEXTOINT function [Data type conversion]

Returns the decimal integer equivalent of a hexadecimal string.

Syntax

HEXTOINT(*hexadecimal-string* **)**

Parameters

hexadecimal-string The string to be converted to an integer.

Remarks

The HEXTOINT function accepts string literals or variables consisting only of digits and the uppercase or lowercase letters A-F, with or without a 0x prefix. The following are all valid uses of HEXTOINT:

```
SELECT HEXTOINT( '0xFFFFFFFF' );
SELECT HEXTOINT( '0x00000100' );
SELECT HEXTOINT( '100' );
SELECT HEXTOINT( '0xffffffff80000001' );
```

The HEXTOINT function removes the 0x prefix, if present. If the data exceeds 8 digits, it must represent a value that can be represented as a signed 32-bit integer value.

The HEXTOINT function returns the platform-independent SQL INTEGER equivalent of the hexadecimal string. The hexadecimal value represents a negative integer if the 8th digit from the right is one of the digits 8–9 and the uppercase or lowercase letters A–F and the previous leading digits are all uppercase or lowercase letter F. The following is not a valid use of HEXTOINT since the argument represents a positive integer value that cannot be represented as a signed 32-bit integer:

```
SELECT HEXTOINT( '0x0080000001' );
```

This function supports NCHAR inputs and/or outputs.

See also

- "INTTOHEX function [Data type conversion]" on page 184

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 420.

```
SELECT HEXTOINT( '1A4' );
```

HOUR function [Date and time]

Returns the hour component of a datetime.

Syntax

HOUR(*datetime-expression* **)**

Parameters

datetime-expression The datetime.

Remarks

The value returned is a number from 0 to 23 corresponding to the datetime hour.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 21:

```
SELECT HOUR( '1998-07-09 21:12:13' );
```

HOURS function [Date and time]

A function that evaluates hours. For specific details, see this function's usage.

Syntax 1: integer

HOURS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

HOURS (*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of hours to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of hours is subtracted from the datetime. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a DATETIME data type.

For information about casting data types, see "CAST function [Data type conversion]" on page 115.

Remarks

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of hours since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two timestamps, this function returns the integer number of hours between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of hours to the specified timestamp. Instead, use the DATEADD function.

See also

- "DATEDIFF function [Date and time]" on page 137
- "DATEADD function [Date and time]" on page 137

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 4, signifying that the second timestamp is four hours after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT HOURS( '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );

SELECT DATEDIFF( hour,
   '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );
```

The following statement returns the value 17517342.

```
SELECT HOURS( '1998-07-13 06:07:12' );
```

The following statements return the datetime 1999-05-13 02:05:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT HOURS(
   CAST( '1999-05-12 21:05:07' AS DATETIME ), 5 );

SELECT DATEADD( hour, 5, '1999-05-12 21:05:07' );
```

HTML_DECODE function [Miscellaneous]

Decodes special character entities that appear in HTML literal strings.

Syntax

HTML_DECODE(*string* **)**

Parameters

string Arbitrary literal string used in an HTML document.

Remarks

This function returns the string argument after making the following set of substitutions:

Characters	Substitution
&quot;	"
&#39;	'
&amp;	&
&lt;	<
&gt;	>
&#x*hexadecimal-number*;	Unicode codepoint, specified as a hexadecimal number. For example, ' returns a single apostrophe.

Characters	Substitution
&#*decimal-number*;	Unicode codepoint, specified as a decimal number. For example, ™ returns the trademark symbol.

When a Unicode codepoint is specified, if the value can be converted to a character in the database character set, it is converted to a character. Otherwise, it is returned uninterpreted.

SQL Anywhere supports all character entity references specified in the HTML 4.01 Specification. See http://www.w3.org/TR/html4/.

See also

- "HTML_ENCODE function [Miscellaneous]" on page 177
- "HTTP_DECODE function [HTTP]" on page 178
- "HTTP_ENCODE function [HTTP]" on page 178

Standards and compatibility

- **SQL/2003** Vendor extension.

HTML_ENCODE function [Miscellaneous]

Encodes special characters within strings to be inserted into HTML documents.

Syntax

HTML_ENCODE(*string* **)**

Parameters

string Arbitrary string to be used in an HTML document.

Remarks

This function returns the string argument after making the following set of substitutions:

Characters	Substitution
"	"
'	'
&	&
<	<
>	>
codes *nn* less than 0x20	&#x*nn*;

This function supports NCHAR inputs and/or outputs.

See also

- "HTML_DECODE function [Miscellaneous]" on page 176
- "HTTP_ENCODE function [HTTP]" on page 178

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_DECODE function [HTTP]

Decodes special characters within strings for use with HTTP.

Syntax

HTTP_DECODE(*string* **)**

Parameters

string Arbitrary string to be used in an HTTP request.

Remarks

This function returns the string argument after replacing all character sequences of the form %*nn*, where *nn* is a hexadecimal value, with the character with code *nn*. In addition, all plus signs (+) are replaced with spaces.

See also

- "HTTP_ENCODE function [HTTP]" on page 178
- "HTML_DECODE function [Miscellaneous]" on page 176

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_ENCODE function [HTTP]

Encodes special characters in strings for use with HTTP.

Syntax

HTTP_ENCODE(*string* **)**

Parameters

string Arbitrary string to be used in an HTTP request.

Remarks

This function returns the string argument after making the following set of substitutions. In addition, all characters with hexadecimal codes less than 1F or greater than 7E are replaced with %*nn*, where *nn* is the character code.

Character	Substitution
space	%20
"	%22
#	%23
%	%25
&	%26
,	%2C
;	%3B
<	%3C
>	%3E
[	%5B
\	%5C
]	%5D
`	%60
{	%7B
|	%7C
}	%7D
character codes *nn* that are less than 0x1f and greater than 0x7f	%*nn*

This function supports NCHAR inputs and/or outputs.

See also

- "HTTP_DECODE function [HTTP]" on page 178
- "HTML_ENCODE function [Miscellaneous]" on page 177

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_HEADER function [HTTP]

Gets the value of an HTTP header.

Syntax

HTTP_HEADER(*header-field-name* **)**

Parameters

header-field-name The name of an HTTP header field.

Remarks

This function returns the value of the named HTTP header field, or NULL if not called from an HTTP service. It is used when processing an HTTP request via a web service.

If a header for the given *header-field-name* does not exist, the return value is NULL. The return value is also NULL when the function is not called from a web service.

Some headers that may be of interest when processing an HTTP web service request include the following. More information on these headers is available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.org/Protocols/rfc2616/rfc2616-sec14.html.

- **Cookie** The cookie value(s), if any, stored by the client, that are associated with the requested URI.
- **Referer** The URL of the page that contained the link to the requested URI.
- **Host** The name or IP of the host that submitted the request.
- **User-Agent** The name of the client application.
- **Accept-Encoding** A list of encodings for the response that are acceptable to the client application.

Three special headers are always defined when processing an HTTP web service request:

- **@HttpMethod** Returns the type of request being processed. Possible values include HEAD, GET, or POST.
- **@HttpURI** The full URI of the request, as it was specified in the HTTP request.
- **@HttpVersion** The HTTP version of the request (for example, 1.0, or 1.1).

These special headers allow access to the first line of a client request (also known as the request line).

See also

- "HTTP_VARIABLE function [HTTP]" on page 181
- "NEXT_HTTP_HEADER function [HTTP]" on page 207
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 208
- "Working with HTTP headers" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example gets the Cookie header value:

```
SET cookie_value = HTTP_HEADER( 'Cookie' );
```

The following example returns the value of the first HTTP header.


```
DECLARE header_name LONG VARCHAR;
DECLARE header_value LONG VARCHAR;
SET header_name = NEXT_HTTP_HEADER( NULL );
SET header_value = HTTP_HEADER( header_name );
```

HTTP_VARIABLE function [HTTP]

Gets the value of an HTTP variable.

Syntax

HTTP_VARIABLE(*var-name* [[, *instance*] , *http-header-field*] **)**

Parameters

var-name The name of an HTTP variable.

instance If more than one variable has the same name, the instance number of the field instance, or NULL to get the first one. Useful for select lists that permit multiple selections.

http-header-field In a multi-part request, a header field name associated with the named field as specified in *var-name*.

Remarks

This function returns the value of the named HTTP variable. It is used when processing an HTTP request within a web service.

If a header for the given *var-name* does not exist, the return value is NULL.

When the web service request is a POST, and the variable data is posted as multipart/form-data, the HTTP server receives HTTP headers for each individual variable. When the *http-header-field* parameter is specified, the HTTP_VARIABLE function returns the associated multipart/form-data header value from the POST request for the particular variable.

All input data goes through character set translation between the client (for example, a browser) character set, and the character set of the database. However, if @BINARY is specified for *http-header-field*, the variable input value is returned without going through character set translation. This may be useful when receiving binary data, such as image data, from a client.

This function returns NULL when not called from a web service.

See also

- "HTTP_HEADER function [HTTP]" on page 179
- "NEXT_HTTP_HEADER function [HTTP]" on page 207
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 208
- "Working with variables" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statements request the Content-Disposition and Content-Type headers of the image variable:

```
SET v_name = HTTP_VARIABLE( 'image', NULL, 'Content-Disposition' );
SET v_type = HTTP_VARIABLE( 'image', NULL, 'Content-Type' );
```

The following statement requests the value of the image variable in its current character set, that is, without going through character set translation:

```
SET v_image = HTTP_VARIABLE( 'image', NULL, '@BINARY' );
```

IDENTITY function [Miscellaneous]

Generates integer values, starting at 1, for each successive row in a query. Its implementation is identical to that of the NUMBER function.

Syntax

IDENTITY(*expression* **)**

Parameters

expression An expression. The expression is parsed, but is ignored during the execution of the function.

Remarks

For a description of how to use the IDENTITY function, see "NUMBER function [Miscellaneous]" on page 211.

See also

- "NUMBER function [Miscellaneous]" on page 211

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns a sequentially-numbered list of employees.

```
SELECT IDENTITY( 10 ), Surname FROM Employees;
```

IFNULL function [Miscellaneous]

Returns the first non NULL expression.

Syntax

IFNULL(*expression-1*, *expression-2* [, *expression-3*] **)**

Parameters

expression-1 The expression to be evaluated. Its value determines whether *expression-2* or *expression-3* is returned.

expression-2 The return value if *expression-1* is NULL.

expression-3 The return value if *expression-1* is not NULL.

Remarks

If the first expression is the NULL value, then the value of the second expression is returned. If the first expression is not NULL, the value of the third expression is returned. If the first expression is not NULL and there is no third expression, NULL is returned.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value –66.

```
SELECT IFNULL( NULL, -66 );
```

The following statement returns NULL, because the first expression is not NULL and there is no third expression.

```
SELECT IFNULL( -66, -66 );
```

INDEX_ESTIMATE function [Miscellaneous]

This function is the same as the ESTIMATE function, except that it always looks only in an index.

Syntax

INDEX_ESTIMATE(*column-name*, *number*
[, *relation-string*]
)

Parameters

column-name The name of the column that is used in the estimate.

number If *number* is specified, the function returns as a REAL the percentage estimate that the query optimizer uses.

relation-string The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: '=' , '>' , '<' , '>=' , '<=' , '<>' , '!=' , '!<' , and '!>'. The default is '='.

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- "ESTIMATE function [Miscellaneous]" on page 156
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 156

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 89.4736862183.

```
SELECT FIRST ESTIMATE( EmployeeID, 200, '>' )
FROM Employees;
```

INSERTSTR function [String]

Inserts a string into another string at a specified position.

Syntax

INSERTSTR(*integer-expression*, *string-expression-1*, *string-expression-2* **)**

Parameters

integer-expression The position after which the string is to be inserted. Use zero to insert a string at the beginning.

string-expression-1 The string into which the other string is to be inserted.

string-expression-2 The string to be inserted.

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "STUFF function [String]" on page 262
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value backoffice.

```
SELECT INSERTSTR( 0, 'office ', 'back' );
```

INTTOHEX function [Data type conversion]

Returns a string containing the hexadecimal equivalent of an integer.

Syntax

INTTOHEX(*integer-expression* **)**

Parameters

integer-expression The integer to be converted to hexadecimal.

See also

- "HEXTOINT function [Data type conversion]" on page 173

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value 0000009c.

```
SELECT INTTOHEX( 156 );
```

ISDATE function [Data type conversion]

Tests if a string argument can be converted to a date.

Syntax

ISDATE(*string* **)**

Parameters

string The string to be analyzed to determine if the string represents a valid date.

Remarks

If a conversion is possible, the function returns 1; otherwise, 0 is returned. If the argument is NULL, 0 is returned.

This function supports NCHAR inputs and/or outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example imports data from an external file, exports rows which contain invalid values, and copies the remaining rows to a permanent table.

```
CREATE GLOBAL TEMPORARY TABLE MyData(
      person VARCHAR(100),
      birth_date VARCHAR(30),
      height_in_cms VARCHAR(10)
   ) ON COMMIT PRESERVE ROWS;
   LOAD TABLE MyData FROM 'exported.dat';
   UNLOAD
      SELECT * FROM MyData
      WHERE ISDATE( birth_date ) = 0
  OR ISNUMERIC( height_in_cms ) = 0
   TO 'badrows.dat';
   INSERT INTO PermData
      SELECT person, birth_date, height_in_cms
```



```
    FROM MyData
    WHERE ISDATE( birth_date ) = 1
AND ISNUMERIC( height_in_cms ) = 1;
 COMMIT;
 DROP TABLE MyData;
```

ISNULL function [Miscellaneous]

Returns the first non-NULL expression from a list. This function is identical to the COALESCE function.

Syntax

ISNULL(*expression*, *expression* [, …] **)**

Parameters

expression An expression to be tested against NULL.

At least two expressions must be passed into the function, and all expressions must be comparable.

Remarks

The return type for this function depends on the expressions specified. That is, when the database server evaluates the function, it first searches for a data type in which all of the expressions can be compared. When found, the database server compares the expressions and then returns the result (the first non-NULL expression) in the type used for the comparison. If the database server cannot find a common comparison type, an error is returned.

See also

- "COALESCE function [Miscellaneous]" on page 118

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value –66.

```
SELECT ISNULL( NULL ,-66, 55, 45, NULL, 16 );
```

ISNUMERIC function [Miscellaneous]

Determines if a string argument is a valid number.

Syntax

ISNUMERIC(*string* **)**

Parameters

string The string to be analyzed to determine if the string represents a valid number.

Remarks

ISNUMERIC returns 1 when the input string evaluates to a valid integer or floating point number; otherwise it returns 0. The function also returns 0 if the string contains only blanks or is NULL.

Following are values that also cause the ISNUMERIC function to return 0:

- Values that use the letter d or D as the exponent separator. For example, 1d2.
- Special values such as NAN, 0x12, INF, and INFINITY.
- NULL (for example, `SELECT ISNUMERIC( NULL );`)

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example imports data from an external file, exports rows that contain invalid values, and copies the remaining rows to a permanent table. In this example, the ISNUMERIC statement validates that the values in height_in_cms values are numeric.

```
CREATE GLOBAL TEMPORARY TABLE MyData(
      person VARCHAR(100),
      birth_date VARCHAR(30),
      height_in_cms VARCHAR(10)
   ) ON COMMIT PRESERVE ROWS;
   LOAD TABLE MyData FROM 'exported.dat';
   UNLOAD
      SELECT *
      FROM MyData
      WHERE ISDATE( birth_date ) = 0
  OR ISNUMERIC( height_in_cms ) = 0
   TO 'badrows.dat';
   INSERT INTO PermData
      SELECT person, birth_date, height_in_cms
      FROM MyData
      WHERE ISDATE( birth_date ) = 1
  AND ISNUMERIC( height_in_cms ) = 1;
   COMMIT;
   DROP TABLE MyData;
```

LAST_VALUE function [Aggregate]

Returns values from the last row of a window.

Syntax

LAST_VALUE(*expression* [**IGNORE NULLS**] **)**
OVER (*window-spec* **)**

window-spec : see the Remarks section below

Parameters

expression The expression to evaluate. For example, a column name.

Remarks

The LAST_VALUE function allows you to select the last value (according to some ordering) in a table, without having to use a self-join. This is valuable when you want to use the last value as the baseline in calculations.

The LAST_VALUE function takes the last record from the partition after doing the ORDER BY. Then, the *expression* is computed against the last record and results are returned.

If IGNORE NULLS is specified, the last non-NULL value of *expression* is returned. If IGNORE NULLs is not specified, the last value is returned whether or not it is NULL.

The LAST_VALUE function is different from most other aggregate functions in that it can only be used with a window specification.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Window aggregate functions" [*SQL Anywhere Server - SQL Usage*]
- "FIRST_VALUE function [Aggregate]" on page 165

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the salary of each employee, plus the name of the employee with the highest salary in the same department:

```
SELECT GivenName + ' ' + Surname AS employee_name,
       Salary, DepartmentID,
       LAST_VALUE( employee_name ) OVER Salary_Window AS highest_paid
FROM Employees
WINDOW Salary_Window AS ( PARTITION BY DepartmentID ORDER BY Salary
                          RANGE BETWEEN UNBOUNDED PRECEDING
                          AND UNBOUNDED FOLLOWING );
```

The result set below shows that Jose Martinez makes the highest salary in department 500, and Scott Evans makes the highest salary in department 400.

employee_name	Salary	DepartmentID	highest_paid
'Michael Lynch'	24903	500	'Jose Martinez'
'Joseph Barker'	27290	500	'Jose Martinez'
'Sheila Romero'	27500	500	'Jose Martinez'
'Felicia Kuo'	28200	500	'Jose Martinez'

employee_name	Salary	DepartmentID	highest_paid
'Jeannette Bertrand'	29800	500	'Jose Martinez'
'Jane Braun'	34300	500	'Jose Martinez'
'Anthony Rebeiro'	34576	500	'Jose Martinez'
'Charles Crowley'	41700	500	'Jose Martinez'
'Jose Martinez'	55500.8	500	'Jose Martinez'
'Doug Charlton'	28300	400	'Scott Evans'
'Elizabeth Lambert'	29384	400	'Scott Evans'
'Joyce Butterfield'	34011	400	'Scott Evans'
'Robert Nielsen'	34889	400	'Scott Evans'
'Alex Ahmed'	34992	400	'Scott Evans'
'Ruth Wetherby'	35745	400	'Scott Evans'
...	...	...	...

LCASE function [String]

Converts all characters in a string to lowercase. This function is identical the LOWER function.

Syntax

LCASE(*string-expression* **)**

Parameters

string-expression The string to be converted to lowercase.

See also

- “LOWER function [String]” on page 196
- “UCASE function [String]” on page 271
- “UPPER function [String]” on page 273
- “String functions” on page 99

Remarks

The LCASE function is similar to the LOWER function.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value chocolate.

```
SELECT LCASE( 'ChoCOlatE' );
```

LEFT function [String]

Returns a number of characters from the beginning of a string.

Syntax

LEFT(*string-expression*, *integer-expression* **)**

Parameters

string-expression The string.

integer-expression The number of characters to return.

Remarks

If the string contains multibyte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.

You can specify an *integer-expression* that is larger than the value in the column. In this case, the entire value is returned.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value will be described in terms of character length semantics.

See also

- "RIGHT function [String]" on page 238
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the first 5 characters of each Surname value in the Customers table.

```
SELECT LEFT( Surname, 5) FROM Customers;
```

LENGTH function [String]

Returns the number of characters in the specified string.

Syntax

LENGTH(*string-expression* **)**

Parameters

string-expression The string.

Remarks

Use this function to determine the length of a string. For example, specify a column name for *string-expression* to determine the length of values in the column.

If the string contains multibyte characters, and the proper collation is being used, LENGTH returns the number of characters, not the number of bytes. If the string is of data type BINARY, the LENGTH function behaves as the BYTE_LENGTH function.

> **Note**
> You can use the LENGTH function and the CHAR_LENGTH function interchangeably for CHAR, VARCHAR, LONG VARCHAR, and NCHAR data types. However, you must use the LENGTH function for BINARY and bit array data types.

This function supports NCHAR inputs and/or outputs.

See also

- "BYTE_LENGTH function [String]" on page 113
- "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 9.

```
SELECT LENGTH( 'chocolate' );
```

LESSER function [Miscellaneous]

Returns the lesser of two parameter values.

Syntax

LESSER(*expression-1*, *expression-2* **)**

Parameters

expression-1 The first parameter value to be compared.

expression-2 The second parameter value to be compared.

Remarks

If the parameters are equal, the first value is returned.

See also

- "GREATER function [Miscellaneous]" on page 171

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT LESSER( 10, 5 ) FROM dummy;
```

LIST function [Aggregate]

Returns a comma-delimited list of values.

Syntax

LIST(
{ *string-expression* | **DISTINCT** *string-expression* }
[**,** *delimiter-string*]
[**ORDER BY** *order-by-expression*] **)**

Parameters

string-expression A string, usually a column name. For each row, the expression's value is added to the comma-separated list.

DISTINCT string-expression An expression; for example, the name of a column that you are using in the query. For each unique value of that column, the value is added to the comma-separated list.

delimiter-string A delimiter string for the list items. The default setting is a comma. There is no delimiter if a value of NULL or an empty string is supplied. The *delimiter-string* must be a constant.

order-by-expression Order the items returned by the function. There is no comma preceding this argument, which makes it easy to use in the case where no *delimiter-string* is supplied.

Multiple LIST functions in the same query block are not allowed to use different *order-by-expression* arguments.

Remarks

NULL values are not added to the list. LIST (X) returns the concatenation (with delimiters) of all the non-NULL values of X for each row in the group. If there does not exist at least one row in the group with a definite X-value, then LIST(X) returns the empty string.

If both DISTINCT and ORDER BY are supplied, the DISTINCT expression must be the same as the ORDER BY expression.

A LIST function cannot be used as a window function, but it can be used as input to a window function.

This function supports NCHAR inputs and/or outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement returns the value 487 Kennedy Court, 547 School Street.

```
SELECT LIST( Street ) FROM Employees
WHERE GivenName = 'Thomas';
```

The following statement lists employee IDs. Each row in the result set contains a comma-delimited list of employee IDs for a single department.

```
SELECT LIST( EmployeeID )
FROM Employees
GROUP BY DepartmentID;
```

LIST(EmployeeID)
102,105,160,243,247,249,266,278,…
129,195,299,467,641,667,690,856,…
148,390,586,757,879,1293,1336,…
184,207,318,409,591,888,992,1062,…
191,703,750,868,921,1013,1570,…

The following statement sorts the employee IDs by the last name of the employee:

```
SELECT LIST( EmployeeID ORDER BY Surname ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```

Sorted IDs '1751,591,1062,1191,992,888,318,184,1576,207,1684,1643,1607,1740,409,1507'

Sorted IDs
1013,191,750,921,868,1658,…
1751,591,1062,1191,992,888,318,…
1336,879,586,390,757,148,1483,…
1039,129,1142,195,667,1162,902,…
160,105,1250,247,266,249,445,…

The following statement returns semicolon-separated lists. Note the position of the ORDER BY clause and the list separator:

```
SELECT LIST( EmployeeID, ';' ORDER BY Surname ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```


Sorted IDs
1013;191;750;921;868;1658;703;…
1751;591;1062;1191;992;888;318;…
1336;879;586;390;757;148;1483;…
1039;129;1142;195;667;1162;902; …
160;105;1250;247;266;249;445;…

Be sure to distinguish the previous statement from the following statement, which returns comma-separated lists of employee IDs sorted by a compound sort-key of `( Surname, ';' )`:

```
SELECT LIST( EmployeeID ORDER BY Surname, ';' ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```

LOCATE function [String]

Returns the position of one string within another.

Syntax

LOCATE(*string-expression-1*, *string-expression-2* [, *integer-expression*] **)**

Parameters

string-expression-1 The string to be searched.

string-expression-2 The string to be searched for. This string is limited to 255 bytes.

integer-expression The character position in the string to begin the search. The first character is position 1. If the starting offset is negative, the locate function returns the last matching string offset rather than the first. A negative offset indicates how much of the end of the string is to be excluded from the search. The number of bytes excluded is calculated as (–1 * offset) –1.

Remarks

If *integer-expression* is specified, the search starts at that offset into the string.

The first string can be a long string (longer than 255 bytes), but the second is limited to 255 bytes. If a long string is given as the second argument, the function returns a NULL value. If the string is not found, 0 is returned. Searching for a zero-length string will return 1. If any of the arguments are NULL, the result is NULL.

If multibyte characters are used, with the appropriate collation, then the starting position and the return value may be different from the *byte* positions.

This function supports NCHAR inputs and/or outputs.

See also

- "String functions" on page 99
- "CHARINDEX function [String]" on page 117

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 8.

```
SELECT LOCATE(
   'office party this week – rsvp as soon as possible',
   'party',
   2 );
```

The following statement:

```
BEGIN
   DECLARE STR LONG VARCHAR;
   DECLARE POS INT;
   SET str = 'c:\test\functions\locate.sql';
   SET pos = LOCATE( str, '\', -1 );
   select str, pos,
      SUBSTR( str, 1, pos -1 ) AS path,
      SUBSTR( str, pos +1 ) AS filename;
END;
```

returns the following output:

str	pos	path	filename
c:\test\functions\locate.sql	18	c:\test\functions	locate.sql

LOG function [Numeric]

Returns the natural logarithm of a number.

Syntax

LOG(*numeric-expression* **)**

Parameters

numeric-expression The number.

See also

- "LOG10 function [Numeric]" on page 196

Remarks

The argument is an expression that returns the value of any built-in numeric data type.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the natural logarithm of 50.

```
SELECT LOG( 50 );
```

LOG10 function [Numeric]

Returns the base 10 logarithm of a number.

Syntax

LOG10(*numeric-expression* **)**

Parameters

numeric-expression The number.

Remarks

The argument is an expression that returns the value of any built-in numeric data type.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

See also

- "LOG function [Numeric]" on page 195

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the base 10 logarithm for 50.

```
SELECT LOG10( 50 );
```

LOWER function [String]

Converts all characters in a string to lowercase. This function is identical the LCASE function.

Syntax

LOWER(*string-expression* **)**

Parameters

string-expression The string to be converted.

Remarks

The LCASE function is identical to the LOWER function.

See also

- "LCASE function [String]" on page 189
- "UCASE function [String]" on page 271
- "UPPER function [String]" on page 273
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value chocolate.

```
SELECT LOWER( 'chOCOLate' );
```

LTRIM function [String]

Trims leading blanks from a string.

Syntax

LTRIM(*string-expression* **)**

Parameters

string-expression The string to be trimmed.

Remarks

The actual length of the result is the length of the expression minus the number of characters removed. If all of the characters are removed, the result is an empty string.

If the parameter can be null, the result can be null.

If the parameter is null, the result is the null value.

This function supports NCHAR inputs and/or outputs.

See also

- "RTRIM function [String]" on page 242
- "TRIM function [String]" on page 270
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

  The TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING) are supplied by the SQL Anywhere LTRIM and RTRIM functions respectively.

Example

The following statement returns the value Test Message with all leading blanks removed.

```
SELECT LTRIM( '     Test Message' );
```

MAX function [Aggregate]

Returns the maximum *expression* value found in each group of rows.

Syntax 1

MAX(*expression* | **DISTINCT** *expression* **)**

Syntax 2

MAX(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

expression The expression for which the maximum value is to be calculated. This is commonly a column name.

DISTINCT expression Returns the same as MAX(*expression*), and is included for completeness.

Remarks

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

This function supports NCHAR inputs and/or outputs.

See also

- "MIN function [Aggregate]" on page 199

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 138948.000, representing the maximum salary in the Employees table.

```
SELECT MAX( Salary )
FROM Employees;
```


MIN function [Aggregate]

Returns the minimum expression value found in each group of rows.

Syntax 1

MIN(*expression* | **DISTINCT** *expression* **)**

Syntax 2

MIN(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

expression The expression for which the minimum value is to be calculated. This is commonly a column name.

DISTINCT expression Returns the same as MIN(*expression*), and is included for completeness.

Remarks

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

This function supports NCHAR inputs and/or outputs.

See also

- "MAX function [Aggregate]" on page 198

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 24903.000, representing the minimum salary in the Employees table.

```
SELECT MIN( Salary )
FROM Employees;
```

MINUTE function [Date and time]

Returns a minute component of a datetime value.

Syntax

MINUTE(*datetime-expression* **)**

Parameters

datetime-expression The datetime value.

Remarks

The value returned is a number from number from 0 to 59 corresponding to the datetime minute.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 22.

```
SELECT MINUTE( '1998-07-13 12:22:34' );
```

MINUTES function [Date and time]

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of minutes since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two timestamps, this function returns the integer number of minutes between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of minutes to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

MINUTES([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

MINUTES(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of minutes to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of minutes is subtracted from the datetime value. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a DATETIME data type.

Remarks

Since this function returns an integer, overflow can occur when syntax 1 is used with timestamps greater than or equal to 4083-03-23 02:08:00.

See also

- "CAST function [Data type conversion]" on page 115

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 240, signifying that the second timestamp is 240 seconds after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT MINUTES( '1999-07-13 06:07:12',
    '1999-07-13 10:07:12' );

SELECT DATEDIFF( minute,
     '1999-07-13 06:07:12',
     '1999-07-13 10:07:12' );
```

The following statement returns the value 1051040527.

```
SELECT MINUTES( '1998-07-13 06:07:12' );
```

The following statements return the timestamp 1999-05-12 21:10:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT MINUTES( CAST( '1999-05-12 21:05:07'
AS DATETIME ), 5);

SELECT DATEADD( minute, 5, '1999-05-12 21:05:07' );
```

MOD function [Numeric]

Returns the remainder when one whole number is divided by another.

Syntax

MOD(*dividend*, *divisor* **)**

Parameters

dividend The dividend, or numerator of the division.

divisor The divisor, or denominator of the division.

Remarks

Division involving a negative dividend gives a negative or zero result. The sign of the divisor has no effect.

See also

- "REMAINDER function [Numeric]" on page 233

Standards and compatibility

- **SQL/2003** SQL foundation feature outside of core SQL.

Example

The following statement returns the value 2.

```
SELECT MOD( 5, 3 );
```

MONTH function [Date and time]

Returns a month of the given date.

Syntax

MONTH(*date-expression* **)**

Parameters

date-expression A datetime value.

Remarks

The value returned is a number from number from 1 to 12 corresponding to the datetime month.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 7.

```
SELECT MONTH( '1998-07-13' );
```

MONTHNAME function [Date and time]

Returns the name of the month from a date.

Syntax

MONTHNAME(*date-expression* **)**

Parameters

date-expression The datetime value.

Remarks

The MONTHNAME function returns a string, even if the result is numeric, such as 2 for the month of February.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value September.


```
SELECT MONTHNAME( '1998-09-05' );
```

MONTHS function [Date and time]

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of months since 0000-02.

  Note
  0000-02 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two timestamps, this function returns the integer number of months between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of minutes to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

MONTHS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

MONTHS(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of months to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of months is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as a datetime data type.

For information about casting data types, see "CAST function [Data type conversion]" on page 115.

Remarks

The value of MONTHS is calculated from the number of first days of the month between the two dates.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 2, signifying that the second date is two months after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT MONTHS( '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );

SELECT DATEDIFF( month,
   '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );
```


The following statement returns the value 23981.

```
SELECT MONTHS( '1998-07-13 06:07:12' );
```

The following statements return the timestamp 1999-10-12 21:05:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT MONTHS( CAST( '1999-05-12 21:05:07'
AS DATETIME ), 5);

SELECT DATEADD( month, 5, '1999-05-12 21:05:07' );
```

NCHAR function [String]

Returns an NCHAR string containing one character whose Unicode code point is given in the parameter, or NULL if the value is not a valid code point value.

Syntax

NCHAR(*integer* **)**

Parameters

integer The number to be converted to the corresponding Unicode code point.

See also

- “CONNECTION_EXTENDED_PROPERTY function [String]” on page 121
- “TO_NCHAR function [String]” on page 267
- “TO_CHAR function [String]” on page 266
- “UNICODE function [String]” on page 272
- “UNISTR function [String]” on page 272

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the ALEF Arabic letter, which is Unicode code point U+627:

```
SELECT NCHAR( 1575 );
```

NEWID function [Miscellaneous]

Generates a UUID (Universally Unique Identifier) value. A UUID is the same as a GUID (Globally Unique Identifier).

Syntax

NEWID( )

Parameters

There are no parameters associated with the NEWID function.

Remarks

The NEWID function generates a unique identifier value. It can be used in a DEFAULT clause for a column.

UUIDs can be used to uniquely identify rows in a table. The values are generated such that a value produced on one computer will not match that produced on another. Hence, they can also be used as keys in synchronization and replication environments.

UUIDs contain hyphens by default for compatibility with other RDBMSs. You change this by setting the uuid_has_hyphens option to Off.

For more information, see "uuid_has_hyphens option [database]" [*SQL Anywhere Server - Database Administration*].

The NEWID function is non-deterministic; successive calls may return different values. The query optimizer does not cache the results of the NEWID function.

For more information about non-deterministic functions, see "Function caching" [*SQL Anywhere Server - SQL Usage*].

See also

- "The NEWID default" [*SQL Anywhere Server - SQL Usage*]
- "STRTOUUID function [String]" on page 261
- "UUIDTOSTR function [String]" on page 274

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a table named mytab with two columns. Column pk has a unique identifier data type, and assigns the NEWID function as the default value. Column c1 has an integer data type.

```
CREATE TABLE mytab(
   pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
   c1 INT );
```

The following statement returns a unique identifier as a string:

```
SELECT NEWID();
```

For example, the value returned might be 96603324-6FF6-49DE-BF7D-F44C1C7E6856.

NEXT_CONNECTION function [System]

Returns an identifying number for the next connection.

Syntax

NEXT_CONNECTION([*connection-id*] [, *database-id*] **)**

Parameters

connection-id An integer, usually returned from a previous call to NEXT_CONNECTION. If *connection-id* is NULL, NEXT_CONNECTION returns the most recent connection ID.

database-id An integer representing one of the databases on the current server. If you supply no *database-id*, the current database is used. If you supply NULL, then NEXT_CONNECTION returns the next connection regardless of database.

Remarks

NEXT_CONNECTION can be used to enumerate the connections to a database. Connection IDs are generally created in monotonically increasing order. This function returns the next connection ID in reverse order.

To get the connection ID value for the most recent connection, enter NULL as the *connection-id*. To get the subsequent connection, enter the previous return value. The function returns NULL when there are no more connections in the order.

NEXT_CONNECTION is useful if you want to disconnect all the connections created before a specific time. However, because NEXT_CONNECTION returns the connection IDS in reverse order, connections made after the function is started are not returned. If you want to ensure that all connections are disconnected, prevent new connections from being created before you run NEXT_CONNECTION.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns an identifier for the first connection on the current database. The identifier is an integer value like 10.

```
SELECT NEXT_CONNECTION( NULL );
```

The following statement returns a value like 5.

```
SELECT NEXT_CONNECTION( 10 );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* on the current database.

```
SELECT NEXT_CONNECTION( connection-id );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* (regardless of database).

```
SELECT NEXT_CONNECTION( connection-id, NULL );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* on the specified database.

```
SELECT NEXT_CONNECTION( connection-id, database-id );
```

The following call returns the first (earliest) connection (regardless of database).

```
SELECT NEXT_CONNECTION( NULL, NULL );
```


The following call returns the first (earliest) connection on the specified database.

```
SELECT NEXT_CONNECTION( NULL, database-id );
```

NEXT_DATABASE function [System]

Returns an identifying number for a database.

Syntax

NEXT_DATABASE({ **NULL** | *database-id* } **)**

Parameters

database-id An integer that specifies the ID number of the database.

Remarks

The NEXT_DATABASE function is used to enumerate the databases running on a database server. To get the first database pass NULL; to get each subsequent database, pass the previous return value. The function returns NULL when there are no more databases. The database ID numbers are not returned in a particular order, but you can tell the order in which connections were made to the server using the database ID. The first database that connects to the server is assigned the value 0, and for subsequent connections to the server, the database IDs are incremented by a value of 1.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value 0, the first database value.

```
SELECT NEXT_DATABASE( NULL );
```

The following statement returns NULL, indicating that there are no more databases on the server.

```
SELECT NEXT_DATABASE( 0 );
```

NEXT_HTTP_HEADER function [HTTP]

Get the next HTTP header name.

Syntax

NEXT_HTTP_HEADER(*header-name* **)**

Parameters

header-name The name of the previous header. If header-name is NULL, this function returns the name of the first HTTP header.

Remarks

This function iterates over the HTTP headers included within a request and returns the next HTTP header name. Calling it with NULL causes it to return the name of the first header. Subsequent headers are retrieved by passing the function the name of the previous header. This function returns NULL when called with the name of the last header, or when not called from a web service.

Calling this function repeatedly returns all the header fields exactly once, but not necessarily in the order they appear in the HTTP request.

See also

- "HTTP_HEADER function [HTTP]" on page 179
- "HTTP_VARIABLE function [HTTP]" on page 181
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 208
- "Working with HTTP headers" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example gets the name of the first HTTP header.

```
DECLARE header_name LONG VARCHAR;
SET header_name  = NULL;
SET header_name = NEXT_HTTP_HEADER( header_name );
```

NEXT_HTTP_VARIABLE function [HTTP]

Get the next HTTP variable name.

Syntax

NEXT_HTTP_VARIABLE(*var-name* **)**

Parameters

var-name The name of the previous variable. If *var-name* is NULL, this function returns the name of the first HTTP variable.

Remarks

This function iterates over the HTTP variables included within a request. Calling it with NULL causes it to return the name of the first variable. Subsequent variables are retrieved by passing the function the name of the previous variable. This function returns NULL when called with the name of the final variable, when not called from a web service.

Calling this function repeatedly returns all the variables exactly once, but not necessarily in the order they appear in the HTTP request. The variables url or url1, url2, …, url10 are included if URL PATH is set to ON or ELEMENTS, respectively.

See also

- "HTTP_HEADER function [HTTP]" on page 179
- "HTTP_VARIABLE function [HTTP]" on page 181
- "NEXT_HTTP_HEADER function [HTTP]" on page 207
- "Working with variables" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example gets the name of the first HTTP variable.

```
DECLARE variable_name LONG VARCHAR;
SET variable_name = NULL;
SET variable_name = NEXT_HTTP_VARIABLE( variable_name );
```

NEXT_SOAP_HEADER function [SOAP]

Returns the next header key in a SOAP request header.

Syntax

NEXT_SOAP_HEADER(*header-key* **)**

Parameters

header-key The XML local name of the top level XML element for the given header entry.

Remarks

If you specify NULL for the *header-key*, the function returns the header key for the first header entry found in the SOAP header.

This function returns NULL if called with the last *header-key*.

See also

- "SOAP_HEADER function [SOAP]" on page 248
- "sa_set_soap_header system procedure" on page 924
- "Working with SOAP headers" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement gets the first header key found in the SOAP header.

```
SET header_key = NEXT_SOAP_HEADER( NULL );
```


NOW function [Date and time]

Returns the current year, month, day, hour, minute, second, and fraction of a second. The accuracy is limited by the accuracy of the system clock.

Syntax

NOW(*)

Remarks

The information the NOW function returns is equivalent to the information returned by the GETDATE function and the CURRENT TIMESTAMP special value.

See also

- "GETDATE function [Date and time]" on page 169
- "CURRENT TIMESTAMP special value" on page 31

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the current date and time.

```
SELECT NOW( * );
```

NULLIF function [Miscellaneous]

Provides an abbreviated CASE expression by comparing expressions.

Syntax

NULLIF(*expression-1*, *expression-2* **)**

Parameters

expression-1 An expression to be compared.

expression-2 An expression to be compared.

Remarks

NULLIF compares the values of the two expressions.

If the first expression equals the second expression, NULLIF returns NULL.

If the first expression does not equal the second expression, or if the second expression is NULL, NULLIF returns the first expression.

The NULLIF function provides a short way to write some CASE expressions.

See also

- "CASE expressions" on page 17

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value a:

```
SELECT NULLIF( 'a', 'b' );
```

The following statement returns NULL.

```
SELECT NULLIF( 'a', 'a' );
```

NUMBER function [Miscellaneous]

Generates numbers starting at 1 for each successive row in the results of the query. The NUMBER function is primarily intended for use in select lists.

Due to limitations imposed by the NUMBER function (described in the Usage section below), use the "ROW_NUMBER function [Miscellaneous]" on page 240, instead. The ROW_NUMBER function provides the same functionality, but without the limitations of the NUMBER function.

Syntax

NUMBER(*)

Remarks

You can use NUMBER(*) in a select list to provide a sequential numbering of the rows in the result set. NUMBER(*) returns the value of the ANSI row number of each result row. This means that the NUMBER function can return positive or negative values, depending on how the application scrolls through the result set. For insensitive cursors, the value of NUMBER(*) will always be positive because the entire result set is materialized at OPEN.

In addition, the row number may be subject to change for some cursor types. The value is fixed for insensitive cursors and scroll cursors. If there are concurrent updates, it may change for dynamic and sensitive cursors.

A syntax error is generated if you use the NUMBER function in: a DELETE statement, a WHERE clause, a HAVING clause, an ORDER BY clause, a subquery, a query involving aggregation, any constraint, a GROUP BY clause, a DISTINCT clause, a query expression (UNION, EXCEPT, INTERSECT), or a derived table.

NUMBER(*) can be used in a view (subject to the above restrictions), but the view column corresponding to the expression involving NUMBER(*) can be referenced at most once in the query or outer view, and the view cannot participate as a NULL-supplying table in a left outer join or full outer join.

In embedded SQL, care should be exercised when using a cursor that references a query containing a NUMBER(*) function. In particular, this function returns negative numbers when a database cursor is positioned using relative to the end of the cursor (an absolute position with a negative offset).

You can use NUMBER in the right hand side of an assignment in the SET clause of an UPDATE statement. For example, `SET x = NUMBER(*)`.

The NUMBER function can also be used to generate primary keys when using the INSERT from SELECT statement (see "INSERT statement" on page 573), although using an AUTOINCREMENT clause is a preferred mechanism for generating sequential primary keys.

For information about the AUTOINCREMENT clause, see "CREATE TABLE statement" on page 450.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a sequentially-numbered list of departments.

```
SELECT NUMBER( * ), DepartmentName
FROM Departments
WHERE DepartmentID > 5
ORDER BY DepartmentName ;
```

PATINDEX function [String]

Returns an integer representing the starting position of the first occurrence of a pattern in a string.

Syntax

PATINDEX('%*pattern***%'**, *string_expression* **)**

Parameters

pattern The pattern to be searched for. If the leading percent wildcard is omitted, the PATINDEX function returns one (1) if the pattern occurs at the beginning of the string, and zero if not.

The pattern uses the same wildcards as the LIKE comparison. These are as follows:

Wildcard	Matches
_ (underscore)	Any one character
% (percent)	Any string of zero or more characters
[]	Any single character in the specified range or set
[^]	Any single character not in the specified range or set

string-expression The string to be searched for the pattern.

Remarks

The PATINDEX function returns the starting position of the first occurrence of the pattern. If the pattern is not found, it returns zero (0).

See also

- "LIKE search condition" on page 23
- "LOCATE function [String]" on page 194

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT PATINDEX( '%hoco%', 'chocolate' );
```

The following statement returns the value 11.

```
SELECT PATINDEX( '%4_5_', '0a1A 2a3A 4a5A' );
```

PERCENT_RANK function [Ranking]

For any row X, defined by the function's arguments and ORDER BY specification, the PERCENT_RANK function determines the rank of row X - 1, divided by the number of rows in the group. The PERCENT_RANK function returns a decimal value between 0 and 1.

Syntax

PERCENT_RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "CUME_DIST function [Ranking]" on page 135
- "DENSE_RANK function [Ranking]" on page 151
- "RANK function [Ranking]" on page 221

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612

Example

The following example returns a result set that shows the ranking of New York employees' salaries by gender. The results are ranked in descending order using a decimal percentage and are partitioned by gender.

```
SELECT DepartmentID, Surname, Salary, Sex,
PERCENT_RANK() OVER (PARTITION BY Sex
ORDER BY Salary DESC) "Rank"
```



```
FROM Employees
WHERE State IN ('NY');
```

DepartmentID	Surname	Salary	Sex	Rank
200	Martel	55700.000	M	0
100	Guevara	42998.000	M	0.333333333
100	Soo	39075.000	M	0.666666667
400	Ahmed	34992.000	M	1
300	Davidson	57090.000	F	0
400	Blaikie	54900.000	F	0.333333333
100	Whitney	45700.000	F	0.666666667
400	Wetherby	35745.000	F	1

PI function [Numeric]

Returns the numeric value PI.

Syntax

PI(*)

Standards and compatibility

- **SQL/2003** Vendor extension.

Remarks

This function returns a DOUBLE value.

Example

The following statement returns the value 3.141592653…

```
SELECT PI( * );
```

PLAN function [Miscellaneous]

Returns the long plan optimization strategy of a SQL statement, as a string.

Syntax

PLAN(*string-expression*, [*cursor-type*], [*update-status*] **)**

Parameters

string-expression The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE statement.

cursor-type A string. *cursor-type* can be asensitive (default), insensitive, sensitive, or keyset-driven.

update-status A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is exactly the same as READ-WRITE.

See also

- "EXPLANATION function [Miscellaneous]" on page 163
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 169

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100' );
```

This information can help with decisions about indexes to add or how to structure your database for better performance.

The following statement returns a string containing the textual plan for an INSENSITIVE cursor over the query `SELECT * FROM Departments WHERE DepartmentID > 100;`.

```
SELECT PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100',
   'insensitive',
   'read-only' );
```

In Interactive SQL, you can view the plan for any SQL statement on the Plan tab in the Results pane.

POWER function [Numeric]

Calculates one number raised to the power of another.

Syntax

POWER(*numeric-expression-1*, *numeric-expression-2* **)**

Parameters

numeric-expression-1 The base.

numeric-expression-2 The exponent.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If any argument is NULL, the result is a NULL value.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 64.

```
SELECT POWER( 2, 6 );
```

PROPERTY function [System]

Returns the value of the specified server-level property as a string.

Syntax

PROPERTY({ *property-id* | *property-name* } **)**

Parameters

property-id An integer that is the property-number of the server-level property. This number can be determined from the PROPERTY_NUMBER function. The *property-id* is commonly used when looping through a set of properties.

property-name A string giving the name of the database property.

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property.

See also

- "Server-level properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the name of the current database server:

```
SELECT PROPERTY( 'Name' );
```


PROPERTY_DESCRIPTION function [System]

Returns a description of a property.

Syntax

PROPERTY_DESCRIPTION({ *property-id* | *property-name* } **)**

Parameters

property-id An integer that is the property-number of the database property. This number can be determined from the PROPERTY_NUMBER function. The *property-id* is commonly used when looping through a set of properties.

property-name A string giving the name of the database property.

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property.

See also

- "Database Properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the description Number of index insertions.

```
SELECT PROPERTY_DESCRIPTION( 'IndAdd' );
```

PROPERTY_NAME function [System]

Returns the name of the property with the supplied property-number.

Syntax

PROPERTY_NAME(*property-id* **)**

Parameters

property-id The property number of the database property.

See also

- "Understanding database properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the property associated with property number 126. The actual property to which this refers changes from release to release.

```
SELECT PROPERTY_NAME( 126 );
```

PROPERTY_NUMBER function [System]

Returns the property number of the property with the supplied property-name.

Syntax

PROPERTY_NUMBER(*property-name* **)**

Parameters

property-name A property name.

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property. In cases where either property number or property name can be used, it is preferable to use the property name. Always use the PROPERTY_NUMBER function to ensure that the property number is current for the server being used.

See also

- "Understanding database properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns an integer value. The actual value changes from release to release.

```
SELECT PROPERTY_NUMBER( 'PAGESIZE' );
```

QUARTER function [Date and time]

Returns a number indicating the quarter of the year from the supplied date expression.

Syntax

QUARTER(*date-expression* **)**

Parameters

date- expression The date.

Remarks

The quarters are as follows:

Quarter	Period (inclusive)
1	January 1 to March 31
2	April 1 to June 30
3	July 1 to September 30
4	October 1 to December 31

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT QUARTER( '1987/05/02' );
```

RADIANS function [Numeric]

Converts a number from degrees to radians.

Syntax

RADIANS(*numeric-expression* **)**

Parameters

numeric-expression A number, in degrees. This angle is converted to radians.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a value of approximately 0.5236.

```
SELECT RADIANS( 30 );
```

RAND function [Numeric]

Returns a random number in the interval 0 to 1, with an optional seed.

Syntax

RAND([*integer-expression*] **)**

Parameters

integer-expression An optional seed used to create a random number. This argument allows you to create repeatable random number sequences.

Remarks

The RAND function is a multiplicative linear congruential random number generator. See Park and Miller (1988), CACM 31(10), pp. 1192-1201 and Press et al. (1992), Numerical Recipes in C (2nd edition, Chapter 7, pp. 279). The result of calling the RAND function is a pseudo-random number *n* where $0 < n < 1$ (neither 0.0 nor 1.0 can be the result).

When a connection is made to the server, the random number generator seeds an initial value. Each connection is uniquely seeded so that it sees a different random sequence from other connections. You can also specify a seed value (*integer-expression*) as an argument. Normally, you should only do this once before requesting a sequence of random numbers through successive calls to the RAND function. If you initialize the seed value more than once, the sequence is restarted. If you specify the same seed value, the same sequence is generated. Seed values that are close in value generate similar initial sequences, with divergence further out in the sequence.

Never combine the sequence generated from one seed value with the sequence generated from a second seed value, in an attempt to obtain statistically random results. In other words, do not reset the seed value at any time during the generation of a sequence of random values.

The RAND function is treated as a non-deterministic function. The query optimizer does not cache the results of the RAND function.

For more information about non-deterministic functions, see "Function caching" [*SQL Anywhere Server - SQL Usage*].

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements produce eleven random results. Each subsequent call to the RAND function where a seed is not specified continues to produce different results:

```
SELECT RAND( 1 );
SELECT RAND( ), RAND( ), RAND( ), RAND( ), RAND( );
SELECT RAND( ), RAND( ), RAND( ), RAND( ), RAND( );
```

The following example produces two sets of results with identical sequences, since the seed value is specified twice:

```
SELECT RAND( 1 ), RAND( ), RAND( ), RAND( ), RAND( );
SELECT RAND( 1 ), RAND( ), RAND( ), RAND( ), RAND( );
```

The following example produces five results that are near each other in value, and are not random in terms of distribution. For this reason, calling the RAND function more than once with similar seed values is not recommended:

```
SELECT RAND( 1 ), RAND( 2 ), RAND( 3 ), RAND( 4 ), RAND( 5 );
```

The following example produces five identical results, and should be avoided:


```
SELECT RAND( 1 ), RAND( 1 ), RAND( 1 ), RAND( 1 ), RAND( 1 );
```

RANK function [Ranking]

Calculates the value of a rank in a group of values. In the case of ties, the RANK function leaves a gap in the ranking sequence.

Syntax

RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "CUME_DIST function [Ranking]" on page 135
- "DENSE_RANK function [Ranking]" on page 151
- "PERCENT_RANK function [Ranking]" on page 213

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612

Example

The following example provides a rank in descending order of employees' salaries in Utah and New York. Notice that the 7th and 8th employees have an identical salary and therefore share the 7th place ranking. The employee that follows receives the 9th place ranking, which leaves a gap in the ranking sequence (no 8th place ranking).

```
SELECT Surname, Salary, State,
RANK() OVER (ORDER BY Salary DESC) "Rank"
FROM Employees WHERE State IN ('NY','UT')
```

Surname	Salary	State	Rank
Shishov	72995.000	UT	1
Wang	68400.000	UT	2
Cobb	62000.000	UT	3
Morris	61300.000	UT	4

Surname	Salary	State	Rank
Davidson	57090.000	NY	5
Martel	55700.000	NY	6
Blaikie	54900.000	NY	7
Diaz	54900.000	NY	7
Driscoll	48023.690	UT	9
Hildebrand	45829.000	UT	10
Whitney	45700.000	NY	11
...	...	...	...
Lynch	24903.000	UT	19

REGR_AVGX function [Aggregate]

Computes the average of the independent variable of the regression line.

Syntax 1

REGR_AVGX(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_AVGX(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *independent-expression*:

```
AVG( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "AVG function [Aggregate]" on page 107
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGY function [Aggregate]" on page 223

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example calculates the average of the dependent variable, employee age.

```
SELECT REGR_AVGX( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees ;
```

REGR_AVGY function [Aggregate]

Computes the average of the dependent variable of the regression line.

Syntax 1

REGR_AVGY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_AVGY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *dependent-expression*:

```
AVG( x )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGX function [Aggregate]" on page 222
- "AVG function [Aggregate]" on page 107

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example calculates the average of the independent variable, employee salary. This function returns the value 49988.6232.

```
SELECT REGR_AVGY( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_COUNT function [Aggregate]

Returns an integer that represents the number of non-NULL number pairs used to fit the regression line.

Syntax 1

REGR_COUNT(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_COUNT(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function returns a LONG as the result.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_AVGX function [Aggregate]" on page 222
- "COUNT function [Aggregate]" on page 129
- "AVG function [Aggregate]" on page 107
- "SUM function [Aggregate]" on page 264

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns a value that indicates the number of non-NULL pairs that were used to fit the regression line. This function returns the value 75.

```
SELECT REGR_COUNT( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_INTERCEPT function [Aggregate]

Computes the y-intercept of the linear regression line that best fits the dependent and independent variables.

Syntax 1

REGR_INTERCEPT(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_INTERCEPT(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *dependent-expression* and *y* represents the *independent-expression*:

```
AVG( x ) - REGR_SLOPE( x, y ) * AVG( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_SLOPE function [Aggregate]" on page 228
- "AVG function [Aggregate]" on page 107

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 4680.6094936855225.


```
SELECT REGR_INTERCEPT( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_R2 function [Aggregate]

Computes the coefficient of determination (also referred to as *R-squared* or the *goodness of fit* statistic) for the regression line.

Syntax 1

REGR_R2(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_R2(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 0.19379959710325653.

```
SELECT REGR_R2( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SLOPE function [Aggregate]

Computes the slope of the linear regression line fitted to non-NULL pairs.

Syntax 1

REGR_SLOPE(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SLOPE(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *dependent-expression* and *y* represents the *independent-expression*:

```
COVAR_POP( x, y ) / VAR_POP( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223
- "COVAR_POP function [Aggregate]" on page 131
- "VAR_POP function [Aggregate]" on page 275

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 935.3429749445614.

```
SELECT REGR_SLOPE( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SXX function [Aggregate]

Returns the sum of squares of the independent expressions used in a linear regression model. The REGR_SXX function can be used to evaluate the statistical validity of a regression model.

Syntax 1

REGR_SXX(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SXX(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values,

the following computation is then made, where *x* represents the *dependent-expression* and *y* represents the *independent-expression*:

```
REGR_COUNT( x, y ) * VAR_POP( x )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_SXY function [Aggregate]" on page 230
- "REGR_SYY function [Aggregate]" on page 231
- "VAR_POP function [Aggregate]" on page 275

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 5916.4800000000105.

```
SELECT REGR_SXX( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SXY function [Aggregate]

Returns the sum of products of the dependent and independent variables. The REGR_SXY function can be used to evaluate the statistical validity of a regression model.

Syntax 1

REGR_SXY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SXY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *dependent-expression* and *y* represents the *independent-expression*:

```
REGR_COUNT( x, y ) * COVAR_POP( x, y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SYY function [Aggregate]" on page 231

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 5533938.004400015.

```
SELECT REGR_SXY( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SYY function [Aggregate]

Returns values that can evaluate the statistical validity of a regression model.

Syntax 1

REGR_SYY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SYY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Remarks

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *dependent-expression* and *y* represents the *independent-expression*:

```
REGR_COUNT( x, y ) * VAR_POP( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_INTERCEPT function [Aggregate]" on page 225
- "REGR_COUNT function [Aggregate]" on page 224
- "REGR_AVGX function [Aggregate]" on page 222
- "REGR_AVGY function [Aggregate]" on page 223
- "REGR_SLOPE function [Aggregate]" on page 228
- "REGR_SXX function [Aggregate]" on page 229
- "REGR_SXY function [Aggregate]" on page 230

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following example returns the value 26, 708, 672,843.3002.

```
SELECT REGR_SYY( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REMAINDER function [Numeric]

Returns the remainder when one whole number is divided by another.

Syntax

REMAINDER(*dividend*, *divisor* **)**

Parameters

dividend The dividend, or numerator of the division.

divisor The divisor, or denominator of the division.

Remarks

Alternatively, try using the MOD function.

See also

- "MOD function [Numeric]" on page 201

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT REMAINDER( 5, 3 );
```

REPEAT function [String]

Concatenates a string a specified number of times.

Syntax

REPEAT(*string-expression*, *integer-expression* **)**

Parameters

string-expression The string to be repeated.

integer-expression The number of times the string is to be repeated. If *integer-expression* is negative, an empty string is returned.

Remarks

If the actual length of the result string exceeds the maximum for the return type, an error occurs. The result is truncated to the maximum string size allowed.

Alternatively, try using the REPLICATE function.

This function supports NCHAR inputs and/or outputs.

See also

- "REPLICATE function [String]" on page 235
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPEAT( 'repeat', 3 );
```

REPLACE function [String]

Replaces a string with another string, and returns the new results.

Syntax

REPLACE(*original-string*, *search-string*, *replace-string* **)**

Parameters

If any argument is NULL, the function returns NULL.

original-string The string to be searched. This can be any length.

search-string The string to be searched for and replaced with *replace-string*. This string is limited to 255 bytes. If *search-string* is an empty string, the original string is returned unchanged.

replace-string The replacement string, which replaces *search-string*. This can be any length. If *replacement-string* is an empty string, all occurrences of *search-string* are deleted.

Remarks

This function replaces all occurrences.

This function supports NCHAR inputs and/or outputs.

See also

- "SUBSTRING function [String]" on page 262
- "CHARINDEX function [String]" on page 117
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value xx.def.xx.ghi.

```
SELECT REPLACE( 'abc.def.abc.ghi', 'abc', 'xx' );
```

The following statement generates a result set containing ALTER PROCEDURE statements which, when executed, would repair stored procedures that reference a table that has been renamed. (To be useful, the table name must be unique.)

```
SELECT REPLACE(
   REPLACE( proc_defn, 'OldTableName', 'NewTableName' ),
   'CREATE PROCEDURE',
   'ALTER PROCEDURE')
FROM SYS.SYSPROCEDURE
WHERE proc_defn LIKE '%OldTableName%';
```

Use a separator other than the comma for the LIST function:

```
SELECT REPLACE( LIST( table_id ), ',', '--')
FROM  SYS.SYSTAB
WHERE table_id <= 5;
```

REPLICATE function [String]

Concatenates a string a specified number of times.

Syntax

REPLICATE(*string-expression*, *integer-expression* **)**

Parameters

string-expression The string to be repeated.

integer-expression The number of times the string is to be repeated.

Remarks

If the actual length of the result string exceeds the maximum for the return type, an error occurs. The result is truncated to the maximum string size allowed.

Alternatively, try using the REPEAT function.

This function supports NCHAR inputs and/or outputs.

See also

- "REPEAT function [String]" on page 233
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPLICATE( 'repeat', 3 );
```

REVERSE function [String]

Returns the reverse of a character expression.

Syntax

REVERSE(*string-expression* **)**

Parameters

string-expression The string to be reversed.

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value cba.

```
SELECT REVERSE( 'abc' );
```

REWRITE function [Miscellaneous]

Returns a rewritten SELECT, UPDATE, or DELETE statement.

Syntax

REWRITE(*select-statement* [**, 'ANSI'**] **)**

Parameters

select-statement The SQL statement to which the rewrite optimizations are applied to generate the function's results.

Remarks

You can use the REWRITE function without the ANSI argument to help understand how the optimizer generated the access plan for a given query. In particular, you can find how SQL Anywhere has rewritten the conditions in the statement's WHERE, ON, and HAVING clauses, and then determine whether or not applicable indexes exist that can be exploited to improve the request's execution time.

The statement that is returned by REWRITE may not match the semantics of the original statement. This is because several rewrite optimizations introduce internal mechanisms that cannot be translated directly into SQL. For example, the server's use of row identifiers to perform duplicate elimination cannot be translated into SQL.

The rewritten query from the REWRITE function is not intended to be executable. It is a tool for analyzing performance issues by showing what gets passed to the optimizer after the rewrite phase.

There are some rewrite optimizations that are not reflected in the output of REWRITE. They include LIKE optimization, optimization for minimum or maximum functions, upper/lower elimination, and predicate subsumption.

If ANSI is specified, REWRITE returns the ANSI equivalent to the statement. In this case, only the following rewrite optimizations are applied:

- Transact-SQL outer joins are rewritten as ANSI SQL outer joins.
- Duplicate correlation names are eliminated.
- KEY and NATURAL joins are rewritten as ANSI SQL joins.

See also

- "Semantic query transformations" [*SQL Anywhere Server - SQL Usage*]
- "extended_join_syntax option [database]" [*SQL Anywhere Server - Database Administration*]
- "Transact-SQL outer joins (*= or =*)" [*SQL Anywhere Server - SQL Usage*]
- "Key joins" [*SQL Anywhere Server - SQL Usage*]
- "Natural joins" [*SQL Anywhere Server - SQL Usage*]
- "Duplicate correlation names in joins (star joins)" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

In the following example, two rewrite optimizations are performed on a query. The first is the unnesting of the subquery into a join between the Employees and SalesOrders tables. The second optimization simplifies the query by eliminating the primary key - foreign key join between Employees and SalesOrders. Part of this rewrite optimization is to replace the join predicate e.EmployeeID=s.SalesRepresentative with the predicate s.SalesRepresentative IS NOT NULL.

```
SELECT REWRITE( 'SELECT s.ID, s.OrderDate
    FROM SalesOrders s
    WHERE EXISTS ( SELECT *
       FROM Employees e
          WHERE e.EmployeeID = s.SalesRepresentative)' ) FROM dummy;
```

The query returns a single column result set containing the rewritten query:

```
'SELECT s.ID, s.OrderDate FROM SalesOrders s WHERE s.SalesRepresentative IS
NOT NULL';
```

The next example of REWRITE uses the ANSI argument.


```
SELECT REWRITE( 'SELECT DISTINCT s.ID, s.OrderDate, e.GivenName, e.EmployeeID
    FROM SalesOrders s, Employees e
          WHERE e.EmployeeID *= s.SalesRepresentative', 'ANSI' ) FROM dummy;
```

The result is the ANSI equivalent of the statement. In this case, the Transact-SQL outer join is converted to an ANSI outer join. The query returns a single column result set (broken into separate lines for readability):

```
'SELECT DISTINCT s.ID, s.OrderDate, e.EmployeeID, e.GivenName
 FROM Employees as e
 LEFT OUTER JOIN  SalesOrders as s
 ON e.EmployeeID = s.SalesRepresentative';
```

RIGHT function [String]

Returns the rightmost characters of a string.

Syntax

RIGHT(*string-expression*, *integer-expression* **)**

Parameters

string-expression The string to be left-truncated.

integer-expression The number of characters at the end of the string to return.

Remarks

If the string contains multibyte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.

You can specify an *integer-expression* that is larger than the value in the column. In this case, the entire value is returned.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value will be described in terms of character length semantics.

See also

- "LEFT function [String]" on page 190
- "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the last 5 characters of each Surname value in the Customers table.

```
SELECT RIGHT( Surname, 5) FROM Customers;
```


ROUND function [Numeric]

Rounds the *numeric-expression* to the specified *integer-expression* amount of places after the decimal point.

Syntax

ROUND(*numeric-expression*, *integer-expression* **)**

Parameters

numeric-expression The number, passed into the function, to be rounded.

integer-expression A positive integer specifies the number of significant digits to the right of the decimal point at which to round. A negative expression specifies the number of significant digits to the left of the decimal point at which to round.

Remarks

The result of this function is either numeric or double. When there is a numeric result and the integer *integer-expression* is a negative value, the precision is increased by one.

See also

- "TRUNCNUM function [Numeric]" on page 270

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 123.200.

```
SELECT ROUND( 123.234, 1 );
```

ROWID function [Miscellaneous]

Returns an unsigned 64-bit value that uniquely identifies a row within a table.

Syntax

ROWID(*correlation-name* **)**

Parameters

correlation-name The correlation name of a table used in the query. The correlation name should refer to a base table, a temporary table, a global temporary table or a proxy table (permitted only when the underlying proxy server supports a similar function). The argument of the ROWID function should not refer to a view, derived table, common table expression or a procedure.

Remarks

Returns the row identifier of table corresponding to the given correlation as an unsigned 64-bit value (BIGINT).

The value returned by the function is not necessarily constant between queries as various operations performed on the database may result in changes to the row identifiers of a table. In particular, the REORGANIZE TABLE statement is likely to result in changes to row identifiers. Additionally, row identifiers may be reused after a row has been deleted. Hence users should refrain from using the ROWID function in ordinary situations; retrieval by primary key value should be used instead. It is recommended that ROWID be used only in diagnostic situations.

Although the result of this function is an UNSIGNED BIGINT, the results of most arithmetic operations on this value have no particular meaning. For example, you should not expect that adding one to a row identifier will give you the row identifier of the next row. Moreover, only equality and IN predicates are sargable if they involve the use of ROWID. If necessary, predicates involving ROWID, such as `ROWID( T ) =` *literal*, cast to a 64-bit UNSIGNED INTEGER value. If the conversion cannot be performed a data exception will occur. If the value of *literal* is an invalid row identifier then the comparison predicate evaluates to FALSE.

The ROWID function cannot be used inside a CHECK constraint on either a table or a column, nor can it be used in the COMPUTE expression for a computed column.

See also

- "ROW_NUMBER function [Miscellaneous]" on page 240

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the row identifier of the row in Employee where id is equal to 105:

```
SELECT ROWID( Employees ) FROM Employees WHERE Employees.EmployeeID = 105;
```

The following statement returns a list of the locks on rows in the Employees table along with the contents of those rows:

```
SELECT *
  FROM sa_locks() S JOIN Employees WITH( NOLOCK )
     ON ROWID( Employees ) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

ROW_NUMBER function [Miscellaneous]

Assigns a unique number to each row. Use this function instead of the NUMBER function.

Syntax

ROW_NUMBER( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER

BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "NUMBER function [Miscellaneous]" on page 211
- "ROWID function [Miscellaneous]" on page 239

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612

Example

The following example returns a result set that provides unique row numbers for each of employees in New York and Utah. Because the query is ordered by Salary in descending order, the first row number is given to the employee with the highest salary in the data set. Although two employees have identical salaries, the tie is not resolved because the two employees are assigned unique row numbers.

```
SELECT Surname, Salary, State,
ROW_NUMBER() OVER (ORDER BY Salary DESC) "Rank"
FROM Employees WHERE State IN ('NY','UT');
```

Surname	Salary	State	Rank
Shishov	72995.000	UT	1
Wang	68400.000	UT	2
Cobb	62000.000	UT	3
Morris	61300.000	UT	4
Davidson	57090.000	NY	5
Martel	55700.000	NY	6
Blaikie	54900.000	NY	7
Diaz	54900.000	NY	8
Driscoll	48023.690	UT	9
Hildebrand	45829.000	UT	10
...	...	...	...
Lynch	24903.000	UT	19

RTRIM function [String]

Returns a string with trailing blanks removed.

Syntax

RTRIM(*string-expression* **)**

Parameters

string-expression The string to be trimmed.

Remarks

The actual length of the result is the length of the expression minus the number of characters removed. If all of the characters are removed, the result is an empty string.

If the argument is null, the result is the null value.

This function supports NCHAR inputs and/or outputs.

See also

- "TRIM function [String]" on page 270
- "LTRIM function [String]" on page 197
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

  The TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING) are supplied by the SQL Anywhere LTRIM and RTRIM functions respectively.

Example

The following statement returns the string Test Message, with all trailing blanks removed.

```
SELECT RTRIM( 'Test Message     ' );
```

SECOND function [Date and time]

Returns a second of the given date.

Syntax

SECOND(*datetime-expression* **)**

Parameters

datetime-expression The datetime value.

Remarks

Returns a number from 0 to 59 corresponding to the second component of the given datetime value.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 25.

```
SELECT SECOND( '1998-07-13 21:21:25' );
```

SECONDS function [Date and time]

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of seconds since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two timestamps, this function returns the integer number of seconds between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of seconds to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

SECONDS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

SECONDS(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of seconds to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of minutes is subtracted from the datetime value. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a datetime data type.

See also

- “CAST function [Data type conversion]” on page 115
- “DATEADD function [Date and time]” on page 137
- “DATEDIFF function [Date and time]” on page 137

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 14400, signifying that the second timestamp is 14400 seconds after the first.


```
SELECT SECONDS( '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );

SELECT DATEDIFF( second,
   '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );
```

The following statement returns the value 63062431632.

```
SELECT SECONDS( '1998-07-13 06:07:12' );
```

The following statements return the datetime 1999-05-12 21:05:12.0.

```
SELECT SECONDS( CAST( '1999-05-12 21:05:07'
AS TIMESTAMP ), 5);

SELECT DATEADD( second, 5, '1999-05-12 21:05:07' );
```

SET_BIT function [Bit array]

Set the value of a specific bit in a bit array.

Syntax

SET_BIT([*bit-expression*,]*bit-position* [, *value*]**)**

Parameters

bit-expression The bit array in which to change the bit.

bit-position The position of the bit to be set. This must be an unsigned integer.

value The value to which the bit is to be set.

Remarks

The default value of *bit-expression* is a bit array of length *bit-position*, containing all bits set to 0 (FALSE).

The default value of *value* is 1 (TRUE).

The result is NULL if any parameter is NULL.

The positions in the array are counted from the left side, starting at 1.

See also

- “GET_BIT function [Bit array]” on page 167
- “SET_BITS function [Aggregate]” on page 245
- “INTEGER data type” on page 60
- “Bitwise operators” on page 13
- “sa_get_bits system procedure” on page 869

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 00100011:

```
SELECT SET_BIT( '00110011', 4 , 0);
```

The following statement returns the value 00111011:

```
SELECT SET_BIT( '00110011', 5 , 1);
```

The following statement returns the value 00111011:

```
SELECT SET_BIT( '00110011', 5 );
```

The following statement returns the value 00001:

```
SELECT SET_BIT( 5 );
```

SET_BITS function [Aggregate]

Creates a bit array where specific bits, corresponding to values from a set of rows, are set to 1 (TRUE).

Syntax

SET_BITS(*expression* **)**

Parameters

expression The expression used to determine which bits to set to 1. This is typically a column name.

Remarks

Rows where the specified values are NULL are ignored.

If there are no rows, NULL is returned.

The length of the result is the largest position that was set to 1.

The SET_BITS function is equivalent to, but faster than, the following statement:

```
SELECT BIT_OR( SET_BIT( expression ) )
FROM table;
```

See also

- “Bitwise operators” on page 13
- “GET_BIT function [Bit array]” on page 167
- “SET_BIT function [Bit array]” on page 244
- “sa_get_bits system procedure” on page 869

Standards and compatibility

SQL/2003 Vendor extension.

Example

The following statements return a bit array with the 2nd, 5th, and 10th bits set to 1 (or 0100100001):


```
CREATE TABLE t( r INTEGER );
INSERT INTO t values( 2 );
INSERT INTO t values( 5 );
INSERT INTO t values(10 );
SELECT SET_BITS( r ) FROM t;
```

SHORT_PLAN function [Miscellaneous]

Returns a short description of the UltraLite plan optimization strategy of a SQL statement, as a string. The description is the same as that returned by the EXPLANATION function.

Syntax

SHORT_PLAN(*string-expression* **)**

Remarks

For some queries, the execution plan for UltraLite may differ from the plan selected for SQL Anywhere.

Parameters

string-expression The SQL statement, which is commonly a SELECT statement, but can also be an UPDATE or DELETE statement.

See also

- “PLAN function [Miscellaneous]” on page 214
- “EXPLANATION function [Miscellaneous]” on page 163
- “GRAPHICAL_PLAN function [Miscellaneous]” on page 169

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT EXPLANATION(
   'SELECT * FROM Departments WHERE DepartmentID > 100' );
```

This information can help with decisions about indexes to add or how to structure your database for better performance.

In Interactive SQL, you can view the plan for any SQL statement on the Plan tab in the Results pane.

SIGN function [Numeric]

Returns the sign of a number.

Syntax

SIGN(*numeric-expression* **)**

Parameters

numeric-expression The number for which the sign is to be returned.

Remarks

For negative numbers, the SIGN function returns -1.

For zero, the SIGN function returns 0.

For positive numbers, the SIGN function returns 1.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value -1

```
SELECT SIGN( -550 );
```

SIMILAR function [String]

Returns a number indicating the similarity between two strings.

Syntax

SIMILAR(*string-expression-1*, *string-expression-2* **)**

Parameters

string-expression-1 The first string to be compared.

string-expression-2 The second string to be compared.

Remarks

The function returns an integer between 0 and 100 representing the similarity between the two strings. The result can be interpreted as the percentage of characters matched between the two strings. A value of 100 indicates that the two strings are identical.

This function can be used to correct a list of names (such as customers). Some customers may have been added to the list more than once with slightly different names. Join the table to itself and produce a report of all similarities greater than 90 percent, but less than 100 percent.

The calculation performed for the SIMILAR function is more complex than just the number of characters that match.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 75, indicating that the two values are 75% similar.

```
SELECT SIMILAR( 'toast', 'coast' );
```

SIN function [Numeric]

Returns the sine of a number.

Syntax

SIN(*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Remarks

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The SIN and ASIN functions are inverse operations.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ASIN function [Numeric]" on page 105
- "COS function [Numeric]" on page 128
- "COT function [Numeric]" on page 129
- "TAN function [Numeric]" on page 265

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the SIN value of 0.52.

```
SELECT SIN( 0.52 );
```

SOAP_HEADER function [SOAP]

Returns a SOAP header entry, or an attribute value for a header entry of the SOAP request.

Syntax

SOAP_HEADER(*header-key* [*index*, *header-attribute*] **)**

Parameters

header-key This VARCHAR parameter specifies the XML local name of the top level XML element for a given SOAP header entry.

index This optional INTEGER parameter differentiates between SOAP header fields that have identical names. This can occur when multiple header entries have top level XML elements with the same localname. Usually, such elements have unique namespaces.

header-attribute This optional VARCHAR parameter can be any attribute node within a header entry element, including:

- **@namespace** A special SQL Anywhere attribute used to access the namespace of the given header entry.
- **mustUnderstand** A SOAP 1.1 header entry attribute indicating whether a header entry is mandatory or optional for the recipient to process.
- **encodingStyle** A SOAP 1.1 header entry attribute indicating the encoding style. This attribute may be accessed, but it is not used internally by SQL Anywhere.
- **actor** A SOAP 1.1 header entry attribute indicating the intended recipient of a header entry by specifying the recipient's URL.

Remarks

This function may be used with a single parameter *header-key* to return a header entry. A header entry is an XML string representation of an element, and all its sub-elements, contained within a SOAP header.

This function may also be used to extract a header entry attribute by specifying the optional *index* and *header-attribute* parameters.

This function returns the value of the named SOAP header field, or NULL if not called from a SOAP service. It is used when processing a SOAP request via a web service.

If a header for the given *header-key* does not exist, the return value is NULL.

See also

- "NEXT_SOAP_HEADER function [SOAP]" on page 209
- "sa_set_soap_header system procedure" on page 924
- "Working with SOAP headers" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

SORTKEY function [String]

Generates sort key values. That is, values that can be used to sort character strings based on alternate collation rules.

Syntax

SORTKEY(*string-expression*
[, { *collation-id*

| *collation-name*[**(***collation-tailoring-string***)**] }]
)

Parameters

string-expression The string expression must contain characters that are encoded in the database's character set.

If *string-expression* is an empty string, the SORTKEY function returns a zero-length binary value. If *string-expression* is NULL, the SORTKEY function returns a NULL value. An empty string has a different sort order value than a NULL string from a database column.

The maximum length of the string that the SORTKEY function can handle is 254 bytes. Any longer part is ignored.

collation-name A string or a character variable that specifies the name of the sort order to use. You can also specify the alias char_collation, or, equivalently, db_collation, to generate sortkeys as used by the CHAR collation in use by the database. Similarly, you can specify the alias nchar_collation to generate sortkeys as used by the NCHAR collation in use by the database.

collation-id A variable, integer constant, or string that specifies the ID number of the sort order to use. This parameter applies only to Adaptive Server Enterprise collations, which can be referred to by their corresponding collation ID.

If you do not specify a collation name or collation ID, the default is Default Unicode multilingual.

Valid collations are as follows:

- SQL Anywhere supported collations. Execute `dbinit -l` to see the supported collations, listed by their corresponding label.
- The Adaptive Server Enterprise collations are listed in the table below.

Description	Collation name	Collation ID
Default Unicode multilingual	default	0
CP 850 Alternative: no accent	altnoacc	39
CP 850 Alternative: lowercase first	altdict	45
CP 850 Western European: no case, preference	altnocsp	46
CP 850 Scandinavian dictionary	scandict	47
CP 850 Scandinavian: no case, preference	scannocp	48
GB Pinyin	gbpinyin	n/a
Binary sort	binary	50
Latin-1 English, French, German dictionary	dict	51
Latin-1 English, French, German no case	nocase	52

Description	Collation name	Collation ID
Latin-1 English, French, German no case, preference	nocasep	53
Latin-1 English, French, German no accent	noaccent	54
Latin-1 Spanish dictionary	espdict	55
Latin-1 Spanish no case	espnocs	56
Latin-1 Spanish no accent	espnoac	57
ISO 8859-5 Russian dictionary	rusdict	58
ISO 8859-5 Russian no case	rusnocs	59
ISO 8859-5 Cyrillic dictionary	cyrdict	63
ISO 8859-5 Cyrillic no case	cyrnocs	64
ISO 8859-7 Greek dictionary	elldict	65
ISO 8859-2 Hungarian dictionary	hundict	69
ISO 8859-2 Hungarian no accents	hunnoac	70
ISO 8859-2 Hungarian no case	hunnocs	71
ISO 8859-5 Turkish dictionary	turdict	72
ISO 8859-5 Turkish no accents	turnoac	73
ISO 8859-5 Turkish no case	turnocs	74
CP 874 (TIS 620) Royal Thai dictionary	thaidict	1
ISO 14651 ordering standard	14651	22
Shift-JIS binary order	sjisbin	179
Unicode UTF-8 binary sort	utf8bin	24
EUC JIS binary order	eucjisbn	192
GB2312 binary order	gb2312bn	137
CP932 MS binary order	cp932bin	129
Big5 binary order	big5bin	194
EUC KSC binary order	euckscbn	161

collation-tailoring-string Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of

keyword=value pairs assembled in parentheses, following the collation name. For example, `'UCA (locale=es;case=LowerFirst;accent=respect)'`. The syntax for specifying these options is identical to the syntax defined for the COLLATION clause of the CREATE DATABASE statement. See "Collation tailoring options" on page 376.

Note
All of the collation tailoring options are supported when specifying the UCA collation. For all other collations, only case sensitivity tailoring is supported.

Remarks

The SORTKEY function generates values that can be used to order results based on predefined sort order behavior. This allows you to work with character sort order behaviors that may not be available from the database collation. The returned value is a binary value that contains coded sort order information for the input string that is retained from the SORTKEY function. For example, you can store the values returned by the SORTKEY function in a column with the source character string. When you want to retrieve the character data in the desired order, the SELECT statement only needs to include an ORDER BY clause on the columns that contain the results of running the SORTKEY function.

The SORTKEY function guarantees that the values it returns for a given set of sort order criteria work for the binary comparisons that are performed on VARBINARY data types.

Generating sortkeys for queries can be expensive. As an alternative for frequently requested sortkeys, consider creating a computed column to hold the sortkey values, and then referencing that column in the ORDER BY clause of the query.

The input of the SORTKEY function can generate up to six bytes of sort order information for each input character. The output of the SORTKEY function is of type VARBINARY and has a maximum length of 1024 bytes.

With respect to collation tailoring, full sensitivity is generally the intent when creating sortkeys so when specifying a non-UCA collation, the default tailoring applied is equivalent to `case=Respect`. For example, the following two statements are equivalent:

```
SELECT SORTKEY( 'abc', '1252LATIN1' );
SELECT SORTKEY( 'abc', '1252LATIN1(case=Respect)' );
```

If UCA is specified by itself, the default tailoring applied is equivalent to `'UCA (case=UpperFirst;accent=Respect;punct=Primary)'`.

If a different tailoring is provided in the second parameter to SORTKEY, those settings override the default settings. For example, the following two statements are equivalent:

```
SELECT SORTKEY( 'abc', 'UCA(accent=Ignore)' );
SELECT SORTKEY( 'abc', 'UCA(case=UpperFirst;accent=Ignore;punct=Primary)' );
```

If the database was created without specifying tailoring options (for example, `dbinit -c -zn uca mydb.db`), the following two clauses may generate different sort orders, even if the database collation name is specified for the SORTKEY function:

```
ORDER BY string-expression
ORDER BY SORTKEY( string-expression, database-collation-name )
```


This is because the default tailoring settings used for database creation and for the SORTKEY function are different. To get the same behavior from SORTKEY as for the database collation, either provide a tailoring syntax for *collation-tailoring-string* that matches the settings for the database collation, or specify db_collation for *collation-name*. For example:

```
SORTKEY( expression, 'db_collation' )
```

Note
Sort key values created using a version of SQL Anywhere prior to 10.0.0 do not contain the same values created using version 10.0.0 and higher. This may be a problem for your applications if your pre-10.0.0 database had sort key values stored within it, especially if sort key value comparison is required by your application. You should regenerate any sort key values in your database that were generated using a version of SQL Anywhere prior to 10.0.0.

See also

- "sort_collation option [database]" [*SQL Anywhere Server - Database Administration*]
- "COMPARE function [String]" on page 119
- "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements queries the Employees table and returns the FirstName and Surname of all employees, sorted by the sortkey values for the Surname column using the dict collation (Latin-1, English, French, German dictionary).

```
SELECT Surname, GivenName FROM Employees ORDER BY SORTKEY( Surname, 'dict' );
```

The following example returns the sortkey value for abc, using the UCA collation and tailoring options.

```
SELECT SORTKEY( 'abc', 'UCA(locale=es;case=LowerFirst;accent=respect)' );
```

SOUNDEX function [String]

Returns a number representing the sound of a string.

Syntax

SOUNDEX(*string-expression* **)**

Parameters

string-expression The string to be evaluated.

Remarks

The SOUNDEX function value for a string is based on the first letter and the next three consonants other than H, Y, and W. Vowels in *string-expression* are ignored unless they are the first letter of the string. Doubled letters are counted as one letter. For example, the word apples is based on the letters A, P, L, and S.

Multibyte characters are ignored by the SOUNDEX function.

Although it is not perfect, the SOUNDEX function normally returns the same number for words that sound similar and that start with the same letter.

The SOUNDEX function works best with English words. It is less useful for other languages.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns two identical numbers, 3827, representing the sound of each name.

```
SELECT SOUNDEX( 'Smith' ), SOUNDEX( 'Smythe' );
```

SPACE function [String]

Returns a specified number of spaces.

Syntax

SPACE(*integer-expression* **)**

Parameters

integer-expression The number of spaces to return.

Remarks

If *integer-expression* is negative, a null string is returned.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a string containing 10 spaces.

```
SELECT SPACE( 10 );
```


SQLDIALECT function [Miscellaneous]

Returns either Watcom-SQL or Transact-SQL, to indicate the SQL dialect of a statement.

Syntax

SQLDIALECT(*sql-statement-string* **)**

Parameters

sql-statement-string The SQL statement that the function uses to determine its dialect.

See also

- "TRANSACTSQL function [Miscellaneous]" on page 269
- "WATCOMSQL function [Miscellaneous]" on page 278

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string Transact-SQL.

```
SELECT
   SQLDIALECT( 'SELECT employeeName = Surname FROM Employees' )
FROM dummy;
```

SQLFLAGGER function [Miscellaneous]

Returns the conformity of a given SQL statement to a specified standard.

Syntax

SQLFLAGGER(*sql-standard-string*, *sql-statement-string* **)**

Parameters

sql-standard-string The standard level against which to test conformance. Possible values are the same as for the sql_flagger_error_level database option:

- **SQL:2003/Core** Test for conformance to core SQL/2003 syntax.
- **SQL:2003/Package** Test for conformance to full SQL/2003 syntax.
- **SQL:1999/Core** Test for conformance to core SQL/1999 syntax.
- **SQL:1999/Package** Test for conformance to full SQL/1999 syntax.
- **SQL:1992/Entry** Test for conformance to entry-level SQL/1992 syntax.
- **SQL:1992/Intermediate** Test for conformance to intermediate-level SQL/1992 syntax.
- **SQL:1992/Full** Test for conformance to full-SQL/1992 syntax.
- **Ultralite** Test for conformance to UltraLite.

sql-statement-string The SQL statement to check for conformance.

See also

- "sql_flagger_error_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "SQL preprocessor" [*SQL Anywhere Server - Programming*]
- "sa_ansi_standard_packages system procedure" on page 839

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement shows an example of the message that is returned when a disallowed extension is found:

```
SELECT SQLFLAGGER( 'SQL:2003/Package', 'SELECT top 1 dummy_col FROM sys.dummy
ORDER BY dummy_col' );
```

This statement returns the message `'0AW03 Disallowed language extension detected in syntax near 'top' on line 1'`.

If no disallowed extensions are found, such as in the following example SQL statement, `'00000'` is returned.

```
SELECT SQLFLAGGER( 'SQL:2003/Package', 'SELECT dummy_col FROM sys.dummy' );
```

SQRT function [Numeric]

Returns the square root of a number.

Syntax

SQRT(*numeric-expression* **)**

Parameters

numeric-expression The number for which the square root is to be calculated.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 3.

```
SELECT SQRT( 9 );
```


STDDEV function [Aggregate]

An alias for STDDEV_SAMP. See "STDDEV_SAMP function [Aggregate]" on page 258.

STDDEV_POP function [Aggregate]

Computes the standard deviation of a population consisting of a numeric-expression, as a DOUBLE.

Syntax 1

STDDEV_POP(*numeric-expression* **)**

Syntax 2

STDDEV_POP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

numeric-expression The expression whose population-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The population-based standard deviation (s) is computed according to the following formula:

$s = [(1/N) * SUM(x_i - mean(x))^2]^{1/2}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794…
2000	2	27.050847	15.0270…
…	…	…	…

STDDEV_SAMP function [Aggregate]

Computes the standard deviation of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax 1

STDDEV_SAMP(*numeric-expression* **)**

Syntax 2

STDDEV_SAMP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

numeric-expression The expression whose sample-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The standard deviation (s) is computed according to the following formula, which assumes a normal distribution:

$s = [(1/(N - 1)) * SUM(x_i - mean(x))^2]^{1/2}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing either 0 or 1 rows.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside of core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_SAMP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218…
2000	2	27.050847	15.0696…
…	…	…	…

STR function [String]

Returns the string equivalent of a number.

Syntax

STR(*numeric-expression* [, *length* [, *decimal*]] **)**

Parameters

numeric-expression Any approximate numeric (float, real, or double precision) expression between –1E126 and 1E127.

length The number of characters to be returned (including the decimal point, all digits to the right and left of the decimal point, and blanks). The default is 10.

decimal The number of decimal digits to be returned. The default is 0.

Remarks

If the integer portion of the number cannot fit in the length specified, then the result is a string of the specified length containing all asterisks. For example, the following statement returns ***.

```
SELECT STR( 1234.56, 3 );
```

> **Note**
> The maximum length that is supported is 128. Any length that is not between 1 and 128 yields a result of NULL.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a string of six spaces followed by 1235, for a total of ten characters.

```
SELECT STR( 1234.56 );
```

The following statement returns the result 1234.6.

```
SELECT STR( 1234.56, 6, 1 );
```

STRING function [String]

Concatenates one or more strings into one large string.

Syntax

STRING(*string-expression* [**,** …]**)**

Parameters

string-expression The string to be evaluated.

If only one argument is supplied, it is converted into a single expression. If more than one argument is supplied, they are concatenated into a single string.

Remarks

Numeric or date parameters are converted to strings before concatenation. The STRING function can also be used to convert any single expression to a string by supplying that expression as the only parameter.

If all parameters are NULL, STRING returns NULL. If any parameters are non-NULL, then any NULL parameters are treated as empty strings.

See also

- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value *testing123*.

```
SELECT STRING( 'testing', NULL, 123 );
```

STRTOUUID function [String]

Converts a string value to a unique identifier (UUID or GUID) value.

> **Not needed in newer databases**
> In databases created before version 9.0.2, the UNIQUEIDENTIFIER data type was defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions were needed to convert between binary and string representations of UUID values.
> In databases created using version 9.0.2 or later, the UNIQUEIDENTIFIER data type was changed to a native data type and SQL Anywhere carries out conversions as needed. You do not need to use STRTOUUID and UUIDTOSTR functions with these versions.
> For more information, see "UNIQUEIDENTIFIER data type" on page 75.

Syntax

STRTOUUID(*string-expression* **)**

Parameters

string-expression A string in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*.

Remarks

Converts a string in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx* , where *x* is a hexadecimal digit, to a unique identifier value.

If the string is not a valid UUID string, a conversion error is returned unless the conversion_error option is set to OFF, in which case it returns NULL.

This function is useful for inserting UUID values into a database.

This function supports NCHAR inputs and/or outputs.

See also

- "UUIDTOSTR function [String]" on page 274
- "NEWID function [Miscellaneous]" on page 204
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
CREATE TABLE T1 (
   pk UNIQUEIDENTIFIER PRIMARY KEY, c1 INT );
INSERT INTO T1 ( pk, c1 )
VALUES ( STRTOUUID('12345678-1234-5678-9012-123456789012'), 1 );
```

STUFF function [String]

Deletes a number of characters from one string and replaces them with another string.

Syntax

STUFF(*string-expression-1*, *start*, *length*, *string-expression-2* **)**

Parameters

string-expression-1 The string to be modified by the STUFF function.

start The character position at which to begin deleting characters. The first character in the string is position 1.

length The number of characters to delete.

string-expression-2 The string to be inserted. To delete a portion of a string using the STUFF function, use a replacement string of NULL.

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "INSERTSTR function [String]" on page 184
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value chocolate pie.

```
SELECT STUFF( 'chocolate cake', 11, 4, 'pie' );
```

SUBSTRING function [String]

Returns a substring of a string.

Syntax

{ **SUBSTRING** | **SUBSTR** } **(** *string-expression*, *start*
[, *length*] **)**

Parameters

string-expression The string from which a substring is to be returned.

start The start position of the substring to return, in characters.

length The length of the substring to return, in characters. If *length* is specified, the substring is restricted to that length.

Remarks

The behavior of this function depends on the setting of the ansi_substring database option. When the ansi_substring option is set to On (the default), the behavior of the SUBSTRING function corresponds to ANSI/ISO SQL/2003 behavior. The behavior is as follows:

ansi_substring option setting	start value	length value
On	The first character in the string is at position 1. A negative or zero start offset is treated as if the string were padded on the left with non-characters.	A positive *length* specifies that the substring ends *length* characters to the right of the starting position. A negative *length* returns an error.
Off	The first character in the string is at position 1. A negative starting position specifies a number of characters from the end of the string instead of the beginning. If *start* is zero and *length* is non-negative, a *start* value of 1 is used. If *start* is zero and *length* is negative, a start value of -1 is used.	A positive *length* specifies that the substring ends *length* characters to the right of the starting position. A negative *length* returns at most *length* characters up to, and including, the starting position, from the left of the starting position.

If *string-expression* is of binary data type, the SUBSTRING function behaves as BYTE_SUBSTR.

It is recommended that you avoid using non-positive start offsets or negative lengths with the SUBSTRING function. Where possible, use the LEFT or RIGHT functions instead.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value is described in terms of character length semantics.

See also

- "BYTE_SUBSTR function [String]" on page 114
- "ansi_substring option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "LEFT function [String]" on page 190
- "RIGHT function [String]" on page 238
- "CHARINDEX function [String]" on page 117
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following table shows the values returned by the SUBSTRING function when used in a SELECT statement, with the ansi_substring option set to On and Off.

Example	Result with ansi_substring set to On	Result with ansi_substring set to Off
SUBSTRING('front yard', 1, 4)	fron	fron
SUBSTRING('back yard', 6, 4)	yard	yard
SUBSTR('abcdefgh', 0, -2)	Returns an error	gh
SUBSTR('abcdefgh', -2, 2)	Returns an empty string	gh
SUBSTR('abcdefgh', 2, -2)	Returns an error	ab
SUBSTR('abcdefgh', 2, -4)	Returns an error	ab
SUBSTR('abcdefgh', 2, -1)	Returns an error	b

SUM function [Aggregate]

Returns the total of the specified expression for each group of rows.

Syntax 1

SUM(*expression* | **DISTINCT** *expression* **)**

Syntax 2

SUM(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

expression The object to be summed. This is commonly a column name.

DISTINCT expression Computes the sum of the unique values of *expression* in the input.

Remarks

Rows where the specified expression is NULL are not included.

Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "COUNT function [Aggregate]" on page 129
- "AVG function [Aggregate]" on page 107

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 3749146.740.

```
SELECT SUM( Salary )
FROM Employees;
```

TAN function [Numeric]

Returns the tangent of a number.

Syntax

TAN(*numeric-expression* **)**

Parameters

numeric-expression An angle, in radians.

Remarks

The ATAN and TAN functions are inverse operations.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "COS function [Numeric]" on page 128
- "SIN function [Numeric]" on page 248

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value of the tan of 0.52.

```
SELECT TAN( 0.52 );
```

TEXTPTR function [Text and image]

Returns the 16-byte binary pointer to the first page of the specified text column.

Syntax

TEXTPTR(*column-name* **)**

Parameters

column-name The name of a text column.

Remarks

This function is included for Transact-SQL compatibility.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Use TEXTPTR to locate the text column, copy, associated with au_id 486-29-1786 in the author's blurbs table.

The text pointer is put into a local variable @val and supplied as a parameter to the readtext command, which returns 5 bytes, starting at the second byte (offset of 1).

```
DECLARE @val VARBINARY(16)
SELECT @val = TEXTPTR(copy)
FROM blurbs
WHERE au_id = "486-29-1786"
READTEXT blurbs.copy @val 1 5 ;
```

TO_CHAR function [String]

Converts character data from any supported character set into the CHAR character set for the database.

Syntax

TO_CHAR(*string-expression* [**,** *source-charset-name*] **)**

Parameters

string-expression The string to be converted.

source-charset-name The character set of the string.

Remarks

If *source-charset-name* is specified, then this function is equivalent to:

```
CAST( CSCONVERT( CAST( string-expression AS BINARY ),
 'db_charset', source-charset-name )
   AS CHAR );
```

For more information about db_charset, see "CSCONVERT function [String]" on page 133.

If *source-charset-name* is not specified, then this function is equivalent to:

```
CAST( string-expression AS CHAR );
```


See also

- "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121
- "CSCONVERT function [String]" on page 133
- "NCHAR function [String]" on page 204
- "TO_NCHAR function [String]" on page 267
- "UNICODE function [String]" on page 272
- "UNISTR function [String]" on page 272

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

If you have a BINARY value containing data in the cp850 character set, the following statement converts the data to the CHAR character set and data type:

```
SELECT TO_CHAR( 'cp850_data', 'cp850' );
```

TO_NCHAR function [String]

Converts character data from any supported character set into the NCHAR character set.

Syntax

TO_NCHAR(*string-expression* [**,** *source-charset-name*] **)**

Parameters

string-expression The string to be converted

source-charset-name The character set of the string.

Remarks

If *source-charset-name* is specified then this function is equivalent to:

```
CAST( CSCONVERT( CAST( string-expression AS BINARY ),
 'nchar_charset', source-charset-name )
 AS NCHAR );
```

For more information about nchar_charset, see "CSCONVERT function [String]" on page 133.

If *source-charset-name* is not provided then this function is equivalent to:

```
CAST( string-expression AS NCHAR );
```

See also

- "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121
- "CSCONVERT function [String]" on page 133
- "NCHAR function [String]" on page 204
- "TO_CHAR function [String]" on page 266

- "UNICODE function [String]" on page 272
- "UNISTR function [String]" on page 272

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

If you have a BINARY value containing data in the cp850 character set, the following example to converts the data to the NCHAR character set and data type:

```
SELECT TO_NCHAR( 'cp850_data', 'cp850' );
```

TODAY function [Date and time]

Returns the current date.

Syntax

TODAY(*)

Remarks

Use this syntax in place of the historical CURRENT DATE function.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the current day according to the system clock.

```
SELECT TODAY( * ) ;
SELECT CURRENT DATE
```

TRACEBACK function [Miscellaneous]

Returns a string containing a traceback of the procedures and triggers that were executing when the most recent exception (error) occurred.

Syntax

TRACEBACK(*)

Remarks

This is useful for debugging procedures and triggers

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

To use the traceback function, enter the following after an error occurs while executing a procedure:

```
SELECT TRACEBACK( * )
```

TRACED_PLAN function [Miscellaneous]

This function is used by Sybase Central to generate a graphical plan for a query using tracing data.

Syntax

TRACED_PLAN(*logging_session_id*, *query_id* **)**

Parameters

logging_session_id Combined with *query_id*, this INTEGER parameter identifies a row from the sa_diagnostic_query table for which to generate the plan.

query_id Combined with *logging_session_id*, this INTEGER parameter identifies a row from the sa_diagnostic_query table for which to generate the plan.

Remarks

This function is for use by Sybase Central.

See also

- "sa_diagnostic_query table" on page 743

Standards and compatibility

- **SQL/2003** Vendor extension.

TRANSACTSQL function [Miscellaneous]

Takes a Watcom-SQL statement and rewrites it in the Transact-SQL dialect.

Syntax

TRANSACTSQL(*sql-statement-string* **)**

Parameters

sql-statement-string The SQL statement that the function uses to determine its dialect.

See also

- "SQLDIALECT function [Miscellaneous]" on page 255
- "WATCOMSQL function [Miscellaneous]" on page 278

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string `'SELECT EmployeeName=empl_name FROM Employees'`.

```
SELECT TRANSACTSQL( 'SELECT empl_name as EmployeeName FROM Employees' ) FROM
dummy;
```

TRIM function [String]

Removes leading and trailing blanks from a string.

Syntax

TRIM(*string-expression* **)**

Parameters

string-expression The string to be trimmed.

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "LTRIM function [String]" on page 197
- "RTRIM function [String]" on page 242
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** The TRIM function is a SQL/2003 core feature.

  SQL Anywhere does not support the additional parameters *trim specification* and *trim character*, as defined in SQL/2003. The SQL Anywhere implementation of TRIM corresponds to a TRIM specification of BOTH.

  For the other TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING), SQL Anywhere supplies the LTRIM and RTRIM functions respectively.

Example

The following statement returns the value chocolate with no leading or trailing blanks.

```
SELECT TRIM( '   chocolate   ' );
```

TRUNCNUM function [Numeric]

Truncates a number at a specified number of places after the decimal point.

Syntax

{ **TRUNCNUM** | **"TRUNCATE"** }**(** *numeric-expression*, *integer-expression* **)**

Parameters

numeric-expression The number to be truncated.

integer-expression A positive integer specifies the number of significant digits to the right of the decimal point at which to round. A negative expression specifies the number of significant digits to the left of the decimal point at which to round.

Remarks

You should use the TRUNCNUM function, not the TRUNCATE function, when truncating numbers.

Use of the TRUNCATE statement is not recommended because the word truncate is a keyword, and therefore requires you to either set the quoted_identifier option to OFF, or put quotes around the word TRUNCATE.

See also

- "ROUND function [Numeric]" on page 239
- "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 600.

```
SELECT TRUNCNUM( 655, -2 );
```

The following statement: returns the value 655.340.

```
SELECT TRUNCNUM( 655.348, 2 );
```

UCASE function [String]

Converts all characters in a string to uppercase. This function is identical the UPPER function.

Syntax

UCASE(*string-expression* **)**

Parameters

string-expression The string to be converted to uppercase.

Remarks

The UCASE function is similar to the UPPER function.

See also

- "UPPER function [String]" on page 273
- "LCASE function [String]" on page 189
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value CHOCOLATE.

```
SELECT UCASE( 'ChocoLate' );
```

UNICODE function [String]

Returns an integer containing the Unicode code point of the first character in the string, or NULL if the first character is not a valid encoding.

Syntax

UNICODE(*nchar-string-expression* **)**

Parameters

nchar-string-expression The NCHAR string whose first character is to be converted to an integer.

See also

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121
- "NCHAR function [String]" on page 204
- "TO_CHAR function [String]" on page 266
- "TO_NCHAR function [String]" on page 267
- "UNISTR function [String]" on page 272

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the integer 65536:

```
SELECT UNICODE(UNISTR( '\u010000data' ));
```

UNISTR function [String]

Converts a string containing characters and Unicode escape sequences to an NCHAR string.

Syntax

UNISTR(*string-expression* **)**

Parameters

string-expression The string to be converted.

Remarks

The UNISTR function allows the use of Unicode characters that cannot be represented in the CHAR character set used by the SQL statement. For example, in an English environment, the UNISTR function could be used to include Chinese characters.

The UNISTR function offers similar functionality to the N" constant, however the UNISTR function allows Unicode characters and characters from the CHAR character set, whereas the N" constant only allows characters from the CHAR character set.

The *string-expression* contains characters and Unicode escape sequences. The Unicode escape sequences are of the form \uXXXX or \uXXXXXX, where each X is a hexadecimal digit. The UNISTR function converts each character and each Unicode escape sequence to the corresponding Unicode character.

If a 6-digit Unicode escape sequence is used, its value must not exceed 10FFFF, the largest Unicode code point. A sequence such as \u234567 is not a 6-digit Unicode escape sequence. It is the 4-digit sequence \u2345 followed by the characters 6 and 7.

If two adjacent Unicode escape sequences form a UTF-16 surrogate pair, they are combined into one Unicode character in the output.

See also

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 121
- "NCHAR function [String]" on page 204
- "TO_CHAR function [String]" on page 266
- "TO_NCHAR function [String]" on page 267
- "UNICODE function [String]" on page 272
- "Strings" on page 8

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example returns the string Hello.

```
SELECT UNISTR( 'Hel\u006c\u006F' );
```

The following example combines the UTF-16 surrogate pair D800-DF02 into the Unicode code point 10302.

```
SELECT UNISTR( '\uD800\uDF02' );
```

The example is equivalent to the previous one.

```
SELECT UNISTR( '\u010302' );
```

UPPER function [String]

Converts all characters in a string to uppercase. This function is identical the UCASE function.

Syntax

UPPER(*string-expression* **)**

Parameters

string-expression The string to be converted to uppercase.

Remarks

The UCASE function is similar to the UPPER function.

See also

- "UCASE function [String]" on page 271
- "LCASE function [String]" on page 189
- "LOWER function [String]" on page 196
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value CHOCOLATE.

```
SELECT UPPER( 'ChocoLate' );
```

UUIDTOSTR function [String]

Converts a unique identifier value (UUID, also known as GUID) to a string value.

> **Not needed in newer databases**
> In databases created before version 9.0.2, the UNIQUEIDENTIFIER data type was defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions were needed to convert between binary and string representations of UUID values.
> In databases created using version 9.0.2 or later, the UNIQUEIDENTIFIER data type was changed to a native data type and SQL Anywhere carries out conversions as needed. You do not need to use STRTOUUID and UUIDTOSTR functions with these versions.
> For more information, see "UNIQUEIDENTIFIER data type" on page 75.

Syntax

UUIDTOSTR(*uuid-expression* **)**

Parameters

uuid-expression A unique identifier value.

Remarks

Converts a unique identifier to a string value in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, where x is a hexadecimal digit. If the binary value is not a valid uniqueidentifier, NULL is returned.

This function is useful if you want to view a UUID value.

See also

- "NEWID function [Miscellaneous]" on page 204
- "STRTOUUID function [String]" on page 261
- "String functions" on page 99

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a table mytab with two columns. Column pk has a unique identifier data type, and column c1 has an integer data type. It then inserts two rows with the values 1 and 2 respectively into column c1.

```
CREATE TABLE mytab(
    pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    c1 INT );
INSERT INTO mytab( c1 ) values ( 1 );
INSERT INTO mytab( c1 ) values ( 2 );
```

Executing the following SELECT statement returns all of the data in the newly created table.

```
SELECT * FROM mytab;
```

You will see a two-column, two-row table. The value displayed for column pk will be binary values.

To convert the unique identifier values into a readable format, execute the following command:

```
SELECT UUIDTOSTR(pk), c1 FROM mytab;
```

The UUIDTOSTR function is not needed for databases created with version 9.0.2 or later.

VAR_POP function [Aggregate]

Computes the statistical variance of a population consisting of a numeric-expression, as a DOUBLE.

Syntax 1

VAR_POP(*numeric-expression* **)**

Syntax 2

VAR_POP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

numeric-expression The expression whose population-based variance is calculated over a set of rows. The expression is commonly a column name.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The population-based variance (s^2) of *numeric-expression* (x) is computed according to the following formula:

$s^2 = (1/N) * SUM(x_i - mean(x))^2$

This variance does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 93

Standards and compatibility

- **SQL/2003** SQL foundation feature (T611) outside of core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021…
2000	2	27.050847	225.8109…
…	…	…	…

VAR_SAMP function [Aggregate]

Computes the statistical variance of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax 1

VAR_SAMP(*numeric-expression* **)**

Syntax 2

VAR_SAMP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Usage section below

Parameters

numeric-expression The expression whose sample-based variance is calculated over a set of rows. The expression is commonly a column name.

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The variance (s^2) of *numeric-expression* (x) is computed according to the following formula, which assumes a normal distribution:

$s^2 = (1/(N - 1)) * SUM(x_i - mean(x))^2$

This variance does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing either 0 or 1 rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 719.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 93
- "VARIANCE function [Aggregate]" on page 278

Standards and compatibility

- **SQL/2003** SQL foundation feature outside of core SQL. The VARIANCE syntax is a vendor extension.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_SAMP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158…
2000	2	27.050847	227.0939…

Year	Quarter	Average	Variance
…	…	…	…

VARIANCE function [Aggregate]

An alias for VAR_SAMP. See "VAR_SAMP function [Aggregate]" on page 276.

VAREXISTS function [Miscellaneous]

Returns 1 if a user-defined variable has been created or declared with a given name. Returns 0 if no such variable has been created.

Syntax

VAREXISTS(*variable-name-string* **)**

Parameters

variable-name-string The variable name to be tested, as a string.

See also

- "CREATE VARIABLE statement" on page 469
- "DECLARE statement" on page 477
- "IF statement" on page 563

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following IF statement creates a variable with a name start_time if one is not already created or declared. The variable can then be used safely.

```
IF VAREXISTS( 'start_time' ) = 0 THEN
    CREATE VARIABLE start_time TIMESTAMP;
END IF;
SET start_time = current timestamp;
```

WATCOMSQL function [Miscellaneous]

Takes a Transact-SQL statement and rewrites it in the Watcom-SQL dialect. This can be useful when converting existing Adaptive Server Enterprise stored procedures into Watcom SQL syntax.

Syntax

WATCOMSQL(*sql-statement-string* **)**

Parameters

sql-statement-string The SQL statement that the function uses to determine its dialect.

See also

- "SQLDIALECT function [Miscellaneous]" on page 255
- "TRANSACTSQL function [Miscellaneous]" on page 269

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string `'SELECT empl_name AS EmployeeName FROM Employees'`.

```
SELECT WATCOMSQL( 'SELECT EmployeeName=empl_name FROM Employees' ) FROM dummy;
```

WEEKS function [Date and time]

Given two dates, this function returns the integer number of weeks between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 137 instead for this purpose.

Given a single date, this function returns the number of weeks since 0000-02-29.

Given one date and an integer, it adds the integer number of weeks to the specified date. It is recommended that you use the "DATEADD function [Date and time]" on page 137 instead for this purpose.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

Syntax 1

WEEKS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

WEEKS(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of weeks to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of weeks is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as a datetime data type.

For information about casting data types, see "CAST function [Data type conversion]" on page 115.

Remarks

The difference of two dates in weeks is the number of Sundays between the two dates.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 8, signifying that the second date is eight weeks after the first. It is recommended that you use the second form (DATEDIFF).

```
SELECT WEEKS( '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );

SELECT DATEDIFF( week,
   '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );
```

The following statement returns the value 104270.

```
SELECT WEEKS( '1998-07-13 06:07:12' );
```

The following statements return the timestamp 1999-06-16 21:05:07.0. It is recommended that you use the second form (DATEADD).

```
SELECT WEEKS( CAST( '1999-05-12 21:05:07'
AS TIMESTAMP ), 5);

SELECT DATEADD( week, 5, '1999-05-12 21:05:07' );
```

XMLAGG function [Aggregate]

Generates a forest of XML elements from a collection of XML values.

Syntax

XMLAGG(*value-expression* [**ORDER BY** *order-by-expression*] **)**

Parameters

value-expression An XML value. The content is escaped unless the data type is XML. The *order-by-expression* orders the elements returned by the function.

order-by-expression An expression used to order the XML elements according to the value of this expression.

Remarks

Any values that are NULL are omitted from the result. If all inputs are NULL, or there are no rows, the result is NULL. If you require a well-formed XML document, you must ensure that your query is written so that the generated XML has a single root element.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains XMLAGG.

For an example of a query that uses the XMLAGG function with an ORDER BY clause, see "Using the XMLAGG function" [*SQL Anywhere Server - SQL Usage*].

See also

- "Using the XMLAGG function" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following statement generates an XML document that shows the orders placed by each customer.

```
SELECT XMLELEMENT( NAME "order",
                   XMLATTRIBUTES( ID AS order_id ),
                     ( SELECT XMLAGG(
                         XMLELEMENT(
                           NAME "Products",
                           XMLATTRIBUTES( ProductID, Quantity AS
"quantity_shipped" ) ) )
                        FROM SalesOrderItems soi
                        WHERE soi.ID = so.ID
                     )
                   ) AS products_ordered
FROM SalesOrders so
ORDER BY so.ID;
```

XMLCONCAT function [String]

Produces a forest of XML elements.

Syntax

XMLCONCAT(*xml-value* [**,** ...]**)**

Parameters

xml-value The XML values to be concatenated.

Remarks

Generates a forest of XML elements. In an unparsed XML document, a forest refers to the multiple root nodes within the document. NULL values are omitted from the result. If all the values are NULL, then NULL is returned. The XMLCONCAT function does not check whether the argument has a prolog. If you require a well-formed XML document, you must ensure that your query is written so that a single root element is generated.

Element content is always escaped unless the data type is XML. Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains a XMLCONCAT function.

See also

- "Using the XMLCONCAT function" [*SQL Anywhere Server - SQL Usage*]
- "XMLFOREST function [String]" on page 284
- "String functions" on page 99

Standards and Compatibility

- Part of the SQL/XML draft standard.

Example

The following query generates <CustomerID>, <cust_fname>, and <cust_lname> elements for each customer.

```
SELECT XMLCONCAT( XMLELEMENT ( NAME CustomerID, ID ),
                  XMLELEMENT( NAME cust_fname, GivenName ),
                  XMLELEMENT( NAME cust_lname, Surname )
                ) AS "Customer Information"
FROM Customers
WHERE ID < 120;
```

XMLELEMENT function [String]

Produces an XML element within a query.

Syntax

XMLELEMENT({ **NAME** *element-name-expression* | *string-expression* }
[, **XMLATTRIBUTES (** *attribute-value-expression*
[**AS** *attribute-name*],... **)**
[, *element-content-expression*,...] **)**

Parameters

element-name-expression An identifier. For each row, an XML element with the same name as the identifier is generated.

attribute-value-expression An attribute of the element. This optional argument allows you to specify an attribute value for the generated element. This argument specifies the attribute name and content. If the *attribute-value-expression* is a column name, then the attribute name defaults to the column name. You can change the attribute name by specifying the *attribute-name argument.*

element-content-expression The content of the element. This can be any string expression. You can specify an unlimited number of *element-content-expression* arguments and they are concatenated together. For example, the following SELECT statement returns the value <x>abcdef</x>:

```
SELECT XMLELEMENT( NAME x, 'abc', 'def' );
```

Remarks

NULL element values and NULL attribute values are omitted from the result. The letter case for both element and attribute names is taken from the query.

Element content is always escaped unless the data type is XML. Invalid element and attribute names are also quoted. For example, consider the following statement:

```
SELECT XMLELEMENT('H1', f_get_page_heading() );
```

If the function f_get_page_heading is defined as RETURNS LONG VARCHAR or RETURNS VARCHAR (1000), then the result is HTML encoded:

```
CREATE FUNCTION f_get_page_heading() RETURNS LONG VARCHAR
   BEGIN
      RETURN ('<B>My Heading</B>');
   END;
```


The above SELECT statement returns the following:

```
<H1>&lt;B&gt;My Heading&lt;/B&gt;</H1>
```

If the function is declared as RETURNS XML, then the above SELECT statement returns the following:

```
<H1><B>My Heading</B></H1>
```

For more information about quoting and the XMLELEMENT function, see "Invalid names and SQL/XML" [*SQL Anywhere Server - SQL Usage*].

XMLELEMENT functions can be nested to create a hierarchy. If you want to return different elements at the same level of the document hierarchy, use the XMLFOREST function.

For more information, see "XMLFOREST function [String]" on page 284.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains the XMLELEMENT function.

See also

- "Using the XMLELEMENT function" [*SQL Anywhere Server - SQL Usage*]
- "XMLFOREST function [String]" on page 284
- "String functions" on page 99

Standards and compatibility

- Part of the SQL/XML draft standard.
- Omitting the NAME keyword and using a string expression as the first argument is a vendor extension.

Example

The following example produces an <item_name> element for each product in the result set, where the product name is the content of the element.

```
SELECT ID, XMLELEMENT( NAME item_name, p.Name )
FROM Products p
WHERE ID > 400;
```

The following example returns `<A HREF="http://www.ianywhere.com/" TARGET="_top">iAnywhere website</A>`:

```
SELECT XMLELEMENT(
   'A',
   XMLATTRIBUTES( 'http://www.ianywhere.com/'
      AS "HREF", '_top' AS "TARGET"),
   'iAnywhere website'
);
```

The following example returns `<table><tbody><tr align="center" valign="top"><td>Cell 1 info</td><td>Cell 2 info</td></tr></tbody></table>`:

```
SELECT XMLELEMENT( name "table",
         XMLELEMENT( name "tbody",
           XMLELEMENT( name "tr",
             XMLATTRIBUTES('center' AS "align", 'top' AS "valign"),
```



```
            XMLELEMENT( name "td", 'Cell 1 info' ),
            XMLELEMENT( name "td", 'Cell 2 info' )
          )
        )
      );
```

The following example returns:`'<x>abcdef</x>','<custom_element>abcdef</custom_element>'`

```
CREATE VARIABLE @my_element_name VARCHAR(200);
SET @my_element_name = 'custom_element';
SELECT XMLELEMENT( NAME x, 'abc', 'def' ),
   XMLELEMENT( @my_element_name,'abc', 'def' );
```

XMLFOREST function [String]

Generates a forest of XML elements.

Syntax

XMLFOREST(*element-content-expression* [**AS** *element-name*],... **)**

Parameters

element-content-expression A string. An element is generated for each *element-content-expression* argument that is specified. The *element-content-expression* value becomes the content of the element. For example, if you specify the EmployeeID column from the Employees table for this argument, then an <EmployeeID> element containing an EmployeeID value is generated for each value in the table.

Specify the *element-name* argument if you want to assign a name other than the *element-content-expression* to the element, otherwise the element name defaults to the *element-content-expression* name.

Remarks

Produces a forest of XML elements. In the unparsed XML document, a forest refers to the multiple root nodes within the document. When all of the arguments to the XMLFOREST function are NULL, a NULL value is returned. If only some values are NULL, the NULL values are omitted from the result. Element content is always quoted unless the data type is XML. You cannot specify attributes using the XMLFOREST function. Use the XMLELEMENT function if you want to specify attributes for generated elements.

For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" on page 282.

Element names are escaped unless the data type is XML.

If you require a well-formed XML document, you must ensure that your query is written so that a single root element is generated.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains XMLFOREST.

See also

- "Using the XMLFOREST function" [*SQL Anywhere Server - SQL Usage*]

- "XMLELEMENT function [String]" on page 282
- "XMLCONCAT function [String]" on page 281
- "String functions" on page 99

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following statement produces an XML element for the first and last name of each employee.

```
SELECT EmployeeID,
       XMLFOREST( GivenName, Surname )
       AS "Employee Name"
FROM Employees;
```

XMLGEN function [String]

Generates an XML value based on an XQuery constructor.

Syntax

XMLGEN(*xquery-constructor*, *content-expression* [**AS** *variable-name*],... **)**

Parameters

xquery-constructor An XQuery constructor. The XQuery constructor is an item defined in the XQuery language. It gives a syntax for constructing XML elements based on XQuery expressions. The *xquery-constructor* argument must be a well-formed XML document with one or more variable references. A variable reference is enclosed in curly braces and must be prefixed with a $ and have no surrounding white space. For example:

```
SELECT XMLGEN( '<a>{$x}</a>', 1 AS x );
```

content-expression A variable. You can specify multiple *content-expression* arguments. The optional *variable-name* argument is used to name the variable. For example,

```
SELECT XMLGEN( '<emp EmployeeID="{$EmployeeID}"><StartDate>{$x}</StartDate></
emp>', EmployeeID, StartDate
                AS x )
FROM Employees;
```

Remarks

Computed constructors as defined in the XQuery specification are not supported by the XMLGEN function.

When you execute a query that contains an XMLGEN function, data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format.

Element content is always escaped unless the data type is XML. Illegal XML element and attribute names are also escaped.

For information about escaping and the XMLGEN function, see "Invalid names and SQL/XML" [*SQL Anywhere Server - SQL Usage*].

See also

- "Using the XMLGEN function" [*SQL Anywhere Server - SQL Usage*]
- "String functions" on page 99

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following example generates an <emp> element, as well as <Surname>, <GivenName>, and <StartDate> elements for each employee.

```
SELECT XMLGEN( '<emp EmployeeID="{$EmployeeID}">
                   <Surname>="{$Surname}"</Surname>
                   <GivenName>="{$GivenName}"</GivenName>
                   <StartDate>="{$StartDate}"</StartDate>
                </emp>',
                EmployeeID,
                Surname,
                GivenName,
                StartDate
              ) AS employee_list
FROM Employees;
```

YEAR function [Date and time]

Takes a timestamp value as a parameter and returns the year specified by that timestamp.

Syntax

YEAR(*datetime-expression* **)**

Parameters

datetime-expression A date, time, or timestamp.

Remarks

The value is returned as a short value.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the value 2001.

```
SELECT YEAR( '2001-09-12' );
```

YEARS function [Date and time]

Given two dates, this function returns the integer number of years between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 137 instead for this purpose.

Given one date, it returns the year. It is recommended that you use the "DATEPART function [Date and time]" on page 140 instead for this purpose.

Given one date and an integer, it adds the integer number of years to the specified date. It is recommended that you use the "DATEADD function [Date and time]" on page 137 instead for this purpose.

Syntax 1

YEARS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

YEARS(*datetime-expression*, *integer-expression* **)**

Parameters

datetime-expression A date and time.

integer-expression The number of years to be added to the *datetime-expression.* If *integer-expression* is negative, the appropriate number of years is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as a datetime data type.

For information about casting data types, see "CAST function [Data type conversion]" on page 115.

Remarks

The value of YEARS is calculated from the number of first days of the year between the two dates.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements both return –4.

```
SELECT YEARS( '1998-07-13 06:07:12',
              '1994-03-13 08:07:13' );

SELECT DATEDIFF( year,
   '1998-07-13 06:07:12',
   '1994-03-13 08:07:13' );
```

The following statements return 1998.

```
SELECT YEARS( '1998-07-13 06:07:12' )
SELECT DATEPART( year, '1998-07-13 06:07:12' );
```

The following statements return the given date advanced 300 years.

```
SELECT YEARS( CAST( '1998-07-13 06:07:12' AS TIMESTAMP ), 300 )

SELECT DATEADD( year, 300, '1998-07-13 06:07:12' );
```


YMD function [Date and time]

Returns a date value corresponding to the given year, month, and day of the month. Values are small integers from -32768 to 32767.

Syntax

YMD(
integer-expression1,
integer-expression2,
integer-expression3 **)**

Parameters

integer-expression1 The year.

integer-expression2 The number of the month. If the month is outside the range 1–12, the year is adjusted accordingly.

integer-expression3 The day number. The day can be any integer; the date is adjusted accordingly.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 1998-06-12.

```
SELECT YMD( 1998, 06, 12 );
```

If the values are outside their normal range, the date will adjust accordingly. For example, the following statement returns the value 2000-03-01.

```
SELECT YMD( 1999, 15, 1 );
```

CHAPTER 4

SQL Statements

Contents

Using the SQL statement reference 295
ALLOCATE DESCRIPTOR statement [ESQL] 299
ALTER DATABASE statement 301
ALTER DBSPACE statement 305
ALTER DOMAIN statement 307
ALTER EVENT statement 308
ALTER FUNCTION statement 310
ALTER INDEX statement 311
ALTER MATERIALIZED VIEW statement 313
ALTER PROCEDURE statement 315
ALTER PUBLICATION statement [MobiLink] [SQL Remote] 317
ALTER REMOTE MESSAGE TYPE statement [SQL Remote] 319
ALTER SERVER statement 321
ALTER SERVICE statement 323
ALTER STATISTICS statement 327
ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 328
ALTER SYNCHRONIZATION USER statement [MobiLink] 330
ALTER TABLE statement 332
ALTER TRIGGER statement 341
ALTER VIEW statement 342
ATTACH TRACING statement 344
BACKUP statement 346
BEGIN statement 351
BEGIN TRANSACTION statement [T-SQL] 354
BREAK statement [T-SQL] 356
CALL statement 357
CASE statement 359
CHECKPOINT statement 361
CLEAR statement [Interactive SQL] 362

CLOSE statement [ESQL] [SP] 363
COMMENT statement 365
COMMIT statement 367
CONFIGURE statement [Interactive SQL] 369
CONNECT statement [ESQL] [Interactive SQL] 370
CONTINUE statement [T-SQL] 373
CREATE DATABASE statement 374
CREATE DBSPACE statement 382
CREATE DECRYPTED FILE statement 384
CREATE DOMAIN statement 386
CREATE ENCRYPTED FILE statement 388
CREATE EVENT statement 390
CREATE EXISTING TABLE statement 395
CREATE EXTERNLOGIN statement 397
CREATE FUNCTION statement 399
CREATE INDEX statement 405
CREATE LOCAL TEMPORARY TABLE statement 409
CREATE MATERIALIZED VIEW statement 411
CREATE MESSAGE statement [T-SQL] 413
CREATE PROCEDURE statement 414
CREATE PROCEDURE statement [T-SQL] 425
CREATE PUBLICATION statement [MobiLink] [SQL Remote] 427
CREATE REMOTE MESSAGE TYPE statement [SQL Remote] 431
CREATE SCHEMA statement 433
CREATE SERVER statement 435
CREATE SERVICE statement 438
CREATE STATISTICS statement 442
CREATE SUBSCRIPTION statement [SQL Remote] 443
CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 445
CREATE SYNCHRONIZATION USER statement [MobiLink] 448
CREATE TABLE statement 450
CREATE TRIGGER statement 462
CREATE TRIGGER statement [T-SQL] 468
CREATE VARIABLE statement 469
CREATE VIEW statement 471

DEALLOCATE statement 474
DEALLOCATE DESCRIPTOR statement [ESQL] 475
Declaration section [ESQL] 476
DECLARE statement 477
DECLARE CURSOR statement [ESQL] [SP] 478
DECLARE CURSOR statement [T-SQL] 482
DECLARE LOCAL TEMPORARY TABLE statement 483
DELETE statement 485
DELETE (positioned) statement [ESQL] [SP] 488
DESCRIBE statement [ESQL] 490
DESCRIBE statement [Interactive SQL] 494
DETACH TRACING statement 496
DISCONNECT statement [ESQL] [Interactive SQL] 497
DROP statement 498
DROP CONNECTION statement 500
DROP DATABASE statement 501
DROP EXTERNLOGIN statement 502
DROP PUBLICATION statement [MobiLink] [SQL Remote] 503
DROP REMOTE MESSAGE TYPE statement [SQL Remote] 504
DROP SERVER statement 505
DROP SERVICE statement 506
DROP STATEMENT statement [ESQL] 507
DROP STATISTICS statement 508
DROP SUBSCRIPTION statement [SQL Remote] 509
DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink] 510
DROP SYNCHRONIZATION USER statement [MobiLink] 511
DROP VARIABLE statement 512
EXCEPT statement 513
EXECUTE statement [ESQL] 515
EXECUTE statement [T-SQL] 517
EXECUTE IMMEDIATE statement [SP] 519
EXIT statement [Interactive SQL] 522
EXPLAIN statement [ESQL] 524
FETCH statement [ESQL] [SP] 526
FOR statement 530

FORWARD TO statement 533
FROM clause 535
GET DATA statement [ESQL] 542
GET DESCRIPTOR statement [ESQL] 544
GET OPTION statement [ESQL] 546
GOTO statement [T-SQL] 547
GRANT statement 548
GRANT CONSOLIDATE statement [SQL Remote] 553
GRANT PUBLISH statement [SQL Remote] 555
GRANT REMOTE statement [SQL Remote] 556
GRANT REMOTE DBA statement [MobiLink] [SQL Remote] 558
GROUP BY clause 559
HELP statement [Interactive SQL] 562
IF statement 563
IF statement [T-SQL] 565
INCLUDE statement [ESQL] 567
INPUT statement [Interactive SQL] 568
INSERT statement 573
INSTALL JAVA statement 578
INTERSECT statement 580
LEAVE statement 582
LOAD STATISTICS statement 584
LOAD TABLE statement 585
LOCK TABLE statement 593
LOOP statement 595
MESSAGE statement 597
OPEN statement [ESQL] [SP] 601
OUTPUT statement [Interactive SQL] 604
PARAMETERS statement [Interactive SQL] 608
PASSTHROUGH statement [SQL Remote] 609
PREPARE statement [ESQL] 610
PREPARE TO COMMIT statement 612
PRINT statement [T-SQL] 613
PUT statement [ESQL] 614
RAISERROR statement [T-SQL] 616

READ statement [Interactive SQL] 618
READTEXT statement [T-SQL] 620
REFRESH MATERIALIZED VIEW statement 621
REFRESH TRACING LEVEL statement 623
RELEASE SAVEPOINT statement 625
REMOTE RESET statement [SQL Remote] 626
REMOVE JAVA statement 627
REORGANIZE TABLE statement 628
RESIGNAL statement 630
RESTORE DATABASE statement 631
RESUME statement 633
RETURN statement 634
REVOKE statement 636
REVOKE CONSOLIDATE statement [SQL Remote] 638
REVOKE PUBLISH statement [SQL Remote] 639
REVOKE REMOTE statement [SQL Remote] 640
REVOKE REMOTE DBA statement [SQL Remote] 641
ROLLBACK statement 642
ROLLBACK TO SAVEPOINT statement 643
ROLLBACK TRANSACTION statement [T-SQL] 644
ROLLBACK TRIGGER statement 645
SAVE TRANSACTION statement [T-SQL] 646
SAVEPOINT statement 647
SELECT statement 648
SET statement 656
SET statement [T-SQL] 658
SET CONNECTION statement [Interactive SQL] [ESQL] 661
SET DESCRIPTOR statement [ESQL] 662
SET OPTION statement 664
SET OPTION statement [Interactive SQL] 667
SET REMOTE OPTION statement [SQL Remote] 668
SET SQLCA statement [ESQL] 670
SETUSER statement 671
SIGNAL statement 673
START DATABASE statement 674

START ENGINE statement [Interactive SQL] 676
START JAVA statement 677
START LOGGING statement [Interactive SQL] 678
START SUBSCRIPTION statement [SQL Remote] 679
START SYNCHRONIZATION DELETE statement [MobiLink] 681
STOP DATABASE statement 683
STOP ENGINE statement 684
STOP JAVA statement 685
STOP LOGGING statement [Interactive SQL] 686
STOP SUBSCRIPTION statement [SQL Remote] 687
STOP SYNCHRONIZATION DELETE statement [MobiLink] 688
SYNCHRONIZE SUBSCRIPTION statement [SQL Remote] 689
SYSTEM statement [Interactive SQL] 691
TRIGGER EVENT statement 692
TRUNCATE TABLE statement 693
UNION statement 695
UNLOAD statement 698
UNLOAD TABLE statement 700
UPDATE statement 703
UPDATE (positioned) statement [ESQL] [SP] 708
UPDATE statement [SQL Remote] 710
VALIDATE statement 713
WAITFOR statement 715
WHENEVER statement [ESQL] 717
WHILE statement [T-SQL] 718
WINDOW clause 719
WRITETEXT statement [T-SQL] 722

Using the SQL statement reference

This section describes some conventions used in documenting the SQL statements.

Common elements in SQL syntax

This section lists language elements that are found in the syntax of many SQL statements.

For more information on the elements described here, see “Identifiers” on page 7, “SQL Data Types” on page 47, “Search conditions” on page 20, “SQL Data Types” on page 47, “Expressions” on page 15, or “Strings” on page 8.

- **column-name**
  An identifier that represents the name of a column. See “Identifiers” on page 7.
- **condition**
  An expression that evaluates to TRUE, FALSE, or UNKNOWN. See “Truth value search conditions” on page 26.
- **connection-name**
  A string representing the name of an active connection. See “Introduction to SQL Anywhere database connections” [*SQL Anywhere Server - Database Administration*].
- **data-type**
  A storage data type. See “SQL Data Types” on page 47.
- **expression**
  An expression. A common example of an expression in syntax is a column name. See “Expressions” on page 15.
- **file-name**
  A string containing a file name.
- **hostvar**
  A C language variable, declared as a host variable preceded by a colon. See “Using host variables” [*SQL Anywhere Server - Programming*].
- **indicator-variable**
  A second host variable of type **short int** immediately following a normal host variable. It must also be preceded by a colon. Indicator variables are used to pass NULL values to and from the database. See “Using host variables” [*SQL Anywhere Server - Programming*].
- **materialized-view-name**
  An identifier that represents the name of a materialized view. See “Working with materialized views” [*SQL Anywhere Server - SQL Usage*].
- **number**
  Any sequence of digits followed by an optional decimal part and preceded by an optional negative sign. Optionally, the number can be followed by an E and then an exponent. For example,


```
42
-4.038
.001
3.4e10
1e-10
```

- **owner**
  An identifier representing the user ID who owns a database object. See "Ownership permissions overview" [*SQL Anywhere Server - Database Administration*].
- **query-block**
  A query block is a simple query expression, or a query expression with an ORDER BY clause.
- **query-expression**
  A query expression can be a SELECT, UNION, INTERSECT, or EXCEPT block (that is, a statement that does not contain an ORDER BY, WITH, FOR, FOR XML, or OPTION clause), or any combination of such blocks.
- **role-name**
  An identifier representing the role name of a foreign key. See "Entities and relationships" [*SQL Anywhere Server - SQL Usage*]
- **savepoint-name**
  An identifier that represents the name of a savepoint. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]
- **search-condition**
  A condition that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 20
- **special-value**
  One of the special values described in "Special values" on page 30.
- **statement-label**
  An identifier that represents the label of a loop or compound statement. See "Control statements" [*SQL Anywhere Server - SQL Usage*].
- **string-expression**
  An expression that resolves to a string. See "Expressions" on page 15.
- **table-list**
  A list of table names, which may include correlation names. See "FROM clause" on page 535 and "Key joins" [*SQL Anywhere Server - SQL Usage*].
- **table-name**
  An identifier that represents the name of a table. See "Identifiers" on page 7.
- **userid**
  An identifier representing a user name. See "Identifiers" on page 7.
- **variable-name**
  An identifier that represents a variable name. See "Variables" on page 36.
- **window-name**

An identifier that represents a window name. Used in syntax related to window definition (for example, the WINDOW clause, and window functions such as RANK). See "Identifiers" on page 7.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the SQL statement ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions appear in italics, like the words *owner* and *table-name* in the following example.

  ALTER TABLE [*owner*.]*table-name*

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

  You might also see square brackets around a portions of keywords. For example, the following syntax indicates that you can use either COMMIT TRAN or COMMIT TRANSACTION:

  COMMIT TRAN[**SACTION**] ...

  Likewise, the following syntax indicates that you can use either COMMIT or COMMIT WORK:

  COMMIT [**WORK**]

- **Repeating items** An item that can be repeated is followed by the appropriate list separator and an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, …]

  In this case, you can specify no column constraint, one, or more. If more than one is specified, they must be separated by commas.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces.

[**QUOTES** { **ON** | **OFF** }]

In this case, if the QUOTES option is chosen, one of ON or OFF must be provided. The brackets and braces should not be typed.

Statement applicability indicators

Some statement titles are followed by an indicator in square brackets that indicate where the statement can be used. These indicators are as follows:

- **[ESQL]** The statement is for use in embedded SQL.
- **[Interactive SQL]** The statement can be used only in Interactive SQL.
- **[SP]** The statement is for use in stored procedures, triggers, or batches.
- **[T-SQL]** The statement is implemented for compatibility with Adaptive Server Enterprise. In some cases, the statement cannot be used in stored procedures that are not in Transact-SQL format. In other cases, an alternative statement closer to the SQL/2003 standard is recommended unless Transact-SQL compatibility is an issue.
- **[MobiLink]** The statement is for use only in MobiLink clients.
- **[SQL Remote]** The statement can be used only in SQL Remote.

If two sets of brackets are used, the statement can be used in both environments. For example, [ESQL][SP] means a statement can be used in both embedded SQL and stored procedures.

ALLOCATE DESCRIPTOR statement [ESQL]

Use this statement to allocate space for a SQL descriptor area (SQLDA).

Syntax

ALLOCATE DESCRIPTOR *descriptor-name*
[**WITH MAX** { *integer* | *hostvar* }]

descriptor-name : *string*

Parameters

WITH MAX clause Allows you to specify the number of variables within the descriptor area. The default size is one. You must still call fill_sqlda to allocate space for the actual data items before doing a fetch or any statement that accesses the data within a descriptor area.

Remarks

Allocates space for a descriptor area (SQLDA). You must declare the following in your C code prior to using this statement:

```
struct sqlda * descriptor_name
```

Permissions

None.

Side effects

None.

See also

- "DEALLOCATE DESCRIPTOR statement [ESQL]" on page 475
- "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following sample program includes an example of ALLOCATE DESCRIPTOR statement usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL INCLUDE SQLCA;
#include "sqldef.h"
EXEC SQL BEGIN DECLARE SECTION;
int         x;
short       type;
int         numcols;
char        string[100];
a_SQL_statement_number  stmt = 0;
EXEC SQL END DECLARE SECTION;
int main(int argc, char * argv[]){
   struct sqlda *      sqlda1;
```



```
    if( !db_init( &sqlca ) ) {
       return 1;
    }
    db_string_connect( &sqlca,
    "UID=dba;PWD=sql;DBF=d:\\DB Files\\sample.db");
    EXEC SQL ALLOCATE DESCRIPTOR sqlda1 WITH MAX 25;
    EXEC SQL PREPARE :stmt FROM
       'SELECT * FROM Employees';
    EXEC SQL DECLARE curs CURSOR FOR :stmt;
    EXEC SQL OPEN curs;
    EXEC SQL DESCRIBE :stmt into sqlda1;
    EXEC SQL GET DESCRIPTOR sqlda1 :numcols=COUNT;
    // how many columns?
    if( numcols > 25 ) {
       // reallocate if necessary
       EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
       EXEC SQL ALLOCATE DESCRIPTOR sqlda1
          WITH MAX :numcols;
       EXEC SQL DESCRIBE :stmt into sqlda1;
    }
    type = DT_STRING; // change the type to string
    EXEC SQL SET DESCRIPTOR sqlda1 VALUE 2 TYPE = :type;
    fill_sqlda( sqlda1 );
    // allocate space for the variables
    EXEC SQL FETCH ABSOLUTE 1 curs
       USING DESCRIPTOR sqlda1;
    EXEC SQL GET DESCRIPTOR sqlda1
       VALUE 2 :string = DATA;
    printf("name = %s", string );
    EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
    EXEC SQL CLOSE curs;
    EXEC SQL DROP STATEMENT :stmt;
    db_string_disconnect( &sqlca, "" );
    db_fini( &sqlca );
    return 0;
}
```

ALTER DATABASE statement

Use this statement to upgrade the database, turn jConnect support for a database on or off, calibrate the database, change the transaction and mirror log file names, or force a mirror server to take ownership of a database.

Syntax 1 - Upgrading components or restoring objects

ALTER DATABASE UPGRADE
[**PROCEDURE ON**]
[**JCONNECT** { **ON** | **OFF** }]

Syntax 2 - Performing calibration

ALTER DATABASE {
CALIBRATE [**SERVER**]
| **CALIBRATE DBSPACE** *dbspace-name*
| **CALIBRATE DBSPACE TEMPORARY**
| **CALIBRATE PARALLEL READ**
| **RESTORE DEFAULT CALIBRATION**
}

Syntax 3 - Changing transaction and mirror log names

ALTER DATABASE *dbfile*
ALTER [**TRANSACTION**] **LOG** {
{ **ON** [*log-name*] [**MIRROR** *mirror-name*] | **OFF** }
[**KEY** *key*]

Syntax 4 - Changing ownership of a database

ALTER DATABASE
{ *dbname* **FORCE START**
| **ALTER DATABASE SET PARTNER FAILOVER** }

Parameters

PROCEDURE clause Drop and re-create all dbo- and sys-owned procedures in the database.

JCONNECT clause To allow the Sybase jConnect JDBC driver access to system catalog information, specify JCONNECT ON. This installs the system objects that provide jConnect support. Specify JCONNECT OFF if you want to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information. JCONNECT is ON by default.

CALIBRATE [SERVER] clause Calibrate all dbspaces except for the temporary dbspace. This clause also performs the work done by CALIBRATE PARALLEL READ.

CALIBRATE DBSPACE clause Calibrate the specified dbspace.

CALIBRATE DBSPACE TEMPORARY clause Calibrate the temporary dbspace.

CALIBRATE PARALLEL READ clause Calibrate the parallel I/O capabilities of devices for all dbspace files. The CALIBRATE [SERVER] clause also performs this calibration.

RESTORE DEFAULT CALIBRATION clause Restore the Disk Transfer Time (DTT) model to the built-in default values that are based on typical hardware and configuration settings.

ALTER [TRANSACTION] LOG clause Change the name of the transaction or mirror log file name. If MIRROR *mirror-name* is not specified, the clause sets a file name for a new transaction log. If the database is not currently using a transaction log, it starts using one. If the database is already using a transaction log, it changes to using the new file as its transaction log.

If MIRROR *mirror-name* is specified, the clause sets a file name for a new transaction log mirror. If the database is not currently using a transaction log mirror, it starts using one. If the database is already using a transaction log mirror, it changes to using the new file as its transaction log mirror.

You can also use this clause to turn off the transaction or mirror log. For example, ALTER DATABASE LOG OFF.

KEY clause Specifies the encryption key to use for the transaction or mirror log. When using the ALTER [TRANSACTION] clause on a strongly encrypted database, you must specify the encryption key.

dbname FORCE START clause Forces a database server that is currently acting as the mirror server to take ownership of the database. This statement must be executed while connected to the database on the primary server, and can be executed from within a procedure or event. See "Forcing a database server to become the primary server" [*SQL Anywhere Server - Database Administration*].

SET PARTNER FAILOVER Initiate a database mirroring failover from the primary server to the mirror server. When executed, any existing connections to the database are closed, including the connection that executed the statement; consequently, if the statement is contained in a procedure or event, other statements that follow it may not be executed. See "Initiating failover on the primary server" [*SQL Anywhere Server - Database Administration*].

Remarks

Syntax 1 You can use the ALTER DATABASE UPGRADE statement as an alternative to the Upgrade utility to upgrade or update a database. This applies to maintenance releases as well. After running this statement, you should restart the database. In general, changes in databases between minor versions are limited to additional database options and minor system table and procedure changes. The ALTER DATABASE UPGRADE statement upgrades the system tables to the current version and adds any new database options. If necessary, it also drops and recreates all system procedures. You can force a rebuild of the system procedures by specifying the PROCEDURE ON clause.

You can also use the ALTER DATABASE UPGRADE statement to restore settings and system objects to their original installed state.

Features that require a physical reorganization of the database file are not made available by executing an ALTER DATABASE UPGRADE statement. Such features include index enhancements and changes in data storage. To obtain the benefits of these enhancements, you must unload and reload your database. See "Rebuilding databases" [*SQL Anywhere Server - SQL Usage*].

Back up before upgrading
As with any software, it is recommended that you make a backup of your database before upgrading. See "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*].

To use the Sybase jConnect JDBC driver to access system catalog information, specify JCONNECT ON (the default). If you want to exclude the jConnect system objects, specify JCONNECT OFF. Setting

JCONNECT OFF does not remove jConnect support from a database. Also, you can still use JDBC, as long as you do not access system catalog information. If you subsequently download a more recent version of jConnect, you can upgrade the version in the database by (re)executing the ALTER DATABASE UPGRADE JCONNECT ON statement. See "Installing jConnect system objects into a database" [*SQL Anywhere Server - Programming*].

Syntax 2 Use Syntax 2 to perform recalibration of the I/O cost model used by the optimizer. This updates the Disk Transfer Time (DTT) model, which is a mathematical model of the disk I/O used by the cost model. When you recalibrate the I/O cost model, the database server is unavailable for other use. In addition, it is essential that all other activities on the computer are idle. Recalibrating the database server is an expensive operation and may take some time to complete. It is recommended that you leave the default in place.

When using the CALIBRATE PARALLEL READ clause, parallel calibration is not performed on dbspace files with fewer than 10000 pages. Even though the database server automatically suspends all of its activity during calibration operations, parallel calibration should be done when there are no processes consuming significant resources on the same computer. After calibration, you can retrieve the maximum estimated number of parallel I/O operations allowed on a dbspace file using the IOParallelism extended database property. See "DB_EXTENDED_PROPERTY function [System]" on page 143.

Syntax 3 You can use the ALTER DATABASE statement to change the transaction and mirror log names associated with a database file. These changes are the same as those made by the Transaction Log (dblog) utility. You can execute this statement while connected to the utility database or another database, depending on the setting of the -gu option. If you are changing the transaction or mirror log of an encrypted database, you must specify a key. You cannot stop using the transaction log if the database is using auditing. Once you turn off auditing, you can stop using the transaction log. This syntax is not supported in procedures, triggers, events, or batches.

Syntax 4 Attempting to execute an ALTER DATABASE *dbname* FORCE START statement for a database that is not being mirrored or is currently active and owned by this server results in an error. Also, if the primary server is still connected to the mirror server, an error is given. See "Introduction to database mirroring" [*SQL Anywhere Server - Database Administration*].

Permissions

For Syntax 1 and 2, must have DBA authority, and must be the only connection to the database. ALTER DATABASE UPGRADE is not supported on Windows CE.

For Syntax 3, you must have file permissions on the directories where the transaction log is located, and the database must not be running.

For Syntax 4, you must have the permissions specified by the -gk server option.

Side effects

Automatic commit

See also

- "CREATE DATABASE statement" on page 374
- "Upgrade utility (dbupgrad)" [*SQL Anywhere Server - Database Administration*]
- "CREATE STATISTICS statement" on page 442
- "Transaction Log utility (dblog)" [*SQL Anywhere Server - Database Administration*]
- "DB_EXTENDED_PROPERTY function [System]" on page 143

- "-gu server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example disables jConnect support:

```
ALTER DATABASE UPGRADE JCONNECT OFF;
```

The following example sets the transaction log file name associated with *demo.db* to *newdemo.log*:

```
ALTER DATABASE 'demo.db'
   ALTER LOG ON 'newdemo.log';
```


ALTER DBSPACE statement

Use this statement to pre-allocate space for a dbspace or for the transaction log, or to update the catalog when a dbspace file is renamed or moved.

Syntax

ALTER DBSPACE { *dbspace-name* | **TRANSLOG** | **TEMPORARY** }
{ **ADD** *number* [**PAGES** | **KB** | **MB** | **GB** | **TB**]
| **RENAME** *file-name-string* }

Parameters

TRANSLOG You supply the special dbspace name TRANSLOG to pre-allocate disk space for the transaction log. Pre-allocation improves performance if the transaction log is expected to grow quickly. You may want to use this feature if, for example, you are handling many binary large objects (BLOBs) such as bitmaps.

TEMPORARY You supply the special dbspace name TEMPORARY to add space to temporary dbspaces. When space is added to a temporary dbspace, the additional space materializes in the corresponding temporary file immediately. Pre-allocating space to the temporary dbspace of a database can improve performance during execution complex queries that use large work tables.

ADD clause An ALTER DBSPACE with the ADD clause pre-allocates disk space for a dbspace. It extends the corresponding database file by the specified size, in units of pages, kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB). If you do not specify a unit, PAGES is the default. The page size of a database is fixed when the database is created.

If space is not pre-allocated, database files are extended by about 256 KB at a time for page sizes of 2 KB, 4 KB, and 8 KB, and by about 32 pages for other page sizes, when the space is needed. Pre-allocating space can improve performance for loading large amounts of data and also serves to keep the database files more contiguous within the file system.

You can use this clause to add space to any of the pre-defined dbspaces (SYSTEM, TEMPORARY, TEMP, TRANSLOG, and TRANSLOGMIRROR). See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

RENAME clause If you rename or move a database file other than the main file to a different directory or device, you can use ALTER DBSPACE with the RENAME clause to ensure that SQL Anywhere finds the new file when the database is started. The name change takes effect as follows:

- If the dbspace was already open before the statement was executed (that is, you have not yet renamed the actual file), it remains accessible; however, the name stored in the catalog is updated. After the database is stopped, you must rename the file to match what you provided using the RENAME clause, otherwise the file name won't match the dbspace name in the catalog and the database server is unable to open the dbspace the next time the database is started.
- If the dbspace was not open when the statement was executed, the database server attempts to open it after updating the catalog. If the dbspace can be opened, it becomes accessible. No error is returned if the dbspace cannot be opened.

To determine if a dbspace is open, execute the statement below. If the result is NULL, the dbspace is not open.

```
SELECT DB_EXTENDED_PROPERTY('FileSize','dbspace-name');
```

Using ALTER DBSPACE with RENAME on the main dbspace, SYSTEM, has no effect.

Remarks

Each database is held in one or more files. A dbspace is an additional file with a logical name associated with each database file, and used to hold more data than can be held in the main database file alone. ALTER DBSPACE modifies the main dbspace (also called the root file) or an additional dbspace. The dbspace names for a database are held in the ISYSFILE system table. The main database file has a dbspace name of SYSTEM.

When a multi-file database is started, the start line or ODBC data source description tells SQL Anywhere where to find the main database file. The main database file holds the system tables. SQL Anywhere looks in these system tables to find the location of the other dbspaces, and then opens each of the other dbspaces. You can specify which dbspace new tables are created in by setting the default_dbspace option.

Permissions

Must have DBA authority. Must be the only connection to the database.

Side effects

Automatic commit.

See also

- "CREATE DBSPACE statement" on page 382
- "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*]
- "Working with databases" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example increases the size of the SYSTEM dbspace by 200 pages:

```
ALTER DBSPACE system
ADD 200;
```

The following example increases the size of the SYSTEM dbspace by 400 MB:

```
ALTER DBSPACE system
ADD 400 MB;
```

The following example changes the file name associated with the system_2 dbspace:

```
ALTER DBSPACE system_2
RENAME 'e:\db\dbspace2.db';
```


ALTER DOMAIN statement

Use this statement to rename a user-defined domain or data type.

Syntax

ALTER { **DOMAIN** | **DATATYPE** } *user-type*
RENAME *new-name*

Remarks

When you execute this statement, the name of the user-defined domain or data type is updated in the ISYSUSERTYPE system table.

> **Note**
> Any procedures, triggers, views, or events that refer to the user-defined domain or data type must be recreated, or else they will continue to refer to the old name.

Permissions

Must have DBA authority or be the database user who created the domain.

Side effects

Automatic commit.

See also

- "ISYSFILE system table" on page 728
- "CREATE DOMAIN statement" on page 386
- "Domains" on page 78
- "Using domains" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example renames the Address domain to MailingAddress:

```
ALTER DOMAIN Address RENAME MailingAddress;
```

ALTER EVENT statement

Use this statement to change the definition of an event or its associated handler for automating predefined actions, or to alter the definition of scheduled actions.

Syntax

ALTER EVENT *event-name*
[**AT** { **CONSOLIDATED** | **REMOTE** | **ALL** }]
[**DELETE TYPE** | **TYPE** *event-type*]
{ **WHERE** { *trigger-condition* | **NULL** }
  | { **ADD** | **ALTER** | **DELETE** } **SCHEDULE** *schedule-spec* }
[**ENABLE** | **DISABLE**]
[[**ALTER**] **HANDLER** *compound-statement* | **DELETE HANDLER** }

event-type :
BackupEnd | **"Connect"** | **ConnectFailed** | **DatabaseStart**
| **DBDiskSpace** | **"Disconnect"** | **GlobalAutoincrement** | **GrowDB**
| **GrowLog** | **GrowTemp** | **LogDiskSpace** | **"RAISERROR"**
| **ServerIdle** | **TempDiskSpace**

trigger-condition :
event_condition(*condition-name* **)** { **=** | **<** | **>** | **!=** | **<=** | **>=** } *value*

schedule-spec :
[*schedule-name*]
  { **START TIME** *start-time* | **BETWEEN** *start-time* **AND** *end-time* }
  [**EVERY** *period* { **HOURS** | **MINUTES** | **SECONDS** }]
  [**ON** { **(** *day-of-week*, … **)** | **(** *day-of-month*, … **)** }]
  [**START DATE** *start-date*]

event-name | *schedule-name* : *identifier*

day-of-week : *string*

value | *period* | *day-of-month* : *integer*

start-time | *end-time* : *time*

start-date : *date*

Parameters

AT clause Use this clause to change the specification regarding the databases at which the event is handled.

DELETE TYPE clause Use this clause to remove an association of the event with an event type. For a description of event types, see "Understanding system events" [*SQL Anywhere Server - Database Administration*].

ADD | ALTER | DELETE SCHEDULE clause Use this clause to change the definition of a schedule. Only one schedule can be altered in any one ALTER EVENT statement.

WHERE clause Use this clause to change the trigger condition under which an event is fired. The WHERE NULL option deletes a condition. For descriptions of most of the parameters, see "CREATE EVENT statement" on page 390.

Remarks

This statement allows you to alter an event definition created with CREATE EVENT. Possible uses include the following:

- You can use ALTER EVENT to change an event handler during development.
- You may want to define and test an event handler without a trigger condition or schedule during a development phase, and then add the conditions for execution using ALTER EVENT once the event handler is completed.
- You may want to disable an event handler temporarily by disabling the event.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "BEGIN statement" on page 351
- "CREATE EVENT statement" on page 390

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER FUNCTION statement

Use this statement to modify a function. You must include the entire new function in the ALTER FUNCTION statement.

Syntax 1

ALTER FUNCTION [*owner*.]*function-name function-definition*

function-definition : CREATE FUNCTION syntax

Syntax 2

ALTER FUNCTION [*owner*.]*function-name* **SET HIDDEN**

Remarks

Syntax 1 The ALTER FUNCTION statement is identical in syntax to the CREATE FUNCTION statement except for the first word. Either version of the CREATE FUNCTION statement can be altered.

Existing permissions on the function are maintained, and do not have to be reassigned. If a DROP FUNCTION and CREATE FUNCTION were carried out, execute permissions would have to be reassigned.

Syntax 2 You can use SET HIDDEN to scramble the definition of the associated function and cause it to become unreadable. The function can be unloaded and reloaded into other databases.

This setting is irreversible. If you will need the original source again, you must maintain it outside the database.

If SET HIDDEN is used, debugging using the debugger will not show the function definition, nor will it be available through procedure profiling.

Permissions

Must be the owner of the function or have DBA authority.

Side effects

Automatic commit.

Standards and compatibility

- **SQL/2003** Vendor extension.

See also

- "CREATE FUNCTION statement" on page 399
- "ALTER PROCEDURE statement" on page 315
- "DROP statement" on page 498
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

ALTER INDEX statement

Use this statement to rename an index, primary key, or foreign key, or to change the clustered nature of an index.

Syntax

ALTER { **INDEX** *index-name*
| [**INDEX**] **FOREIGN KEY** *role-name*
| [**INDEX**] **PRIMARY KEY** }
ON [*owner*.]*object-name* { **REBUILD** | *rename-clause* | *cluster-clause* } }

object-name : *table-name* | *materialized-view-name*

rename-clause : **RENAME** { **AS** | **TO** } *new-index-name*

cluster-clause : **CLUSTERED** | **NONCLUSTERED**

Parameters

rename-clause Specify the new name for the index, primary key, or foreign key.

cluster-clause Specify whether the index should be changed to CLUSTERED or NONCLUSTERED. Only one index on a table can be clustered.

REBUILD clause Use this clause to rebuild an index, instead of dropping and recreating it.

Remarks

The ALTER INDEX statement carries out two tasks:

- It can be used to rename an index, primary key, or foreign key.
- It can be used to change an index type from nonclustered to clustered, or vice versa.

  The ALTER INDEX statement can be used to change the clustering specification of the index, but does not reorganize the data. As well, only one index per table or materialized view can be clustered.

ALTER INDEX cannot be used to change an index on a local temporary table. An attempt to do so will result in an Index not found error.

Permissions

Must own the table, or have REFERENCES permissions on the table or materialized view, or have DBA authority.

Cannot be used within a snapshot transaction. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit. Clears the Results tab in the Results pane in Interactive SQL. Closes all cursors for the current connection.

See also

- "CREATE INDEX statement" on page 405

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement renames the index IX_product_name on the Products table to ixProductName:

```
ALTER INDEX IX_product_name ON Products
RENAME TO ixProductName;
```

The following statement changes IX_product_name to be a clustered index:

```
ALTER INDEX IX_product_name ON Products
CLUSTERED;
```


ALTER MATERIALIZED VIEW statement

Use this statement to alter a materialized view.

Syntax

ALTER MATERIALIZED VIEW [*owner*.]*materialized-view-name* {
SET HIDDEN
| { **ENABLE** | **DISABLE** }
| { **ENABLE** | **DISABLE** } **USE IN OPTIMIZATION**
| { **ADD PCTFREE** *percent-free-space* | **DROP PCTFREE** }
| [**NOT**] **ENCRYPTED**
}

percent-free-space : integer

Parameters

SET HIDDEN clause Use the SET HIDDEN clause to obfuscate the definition of the materialized view. *This setting is irreversible.* For more information, see "Hiding materialized views" [*SQL Anywhere Server - SQL Usage*].

ENABLE clause Use the ENABLE clause to enable a materialized view, making it available for use by the database server. This clause has no effect on a view that is already enabled. After using this clause, you must execute a REFRESH MATERIALIZED VIEW statement to initialize the materialized view with data.

DISABLE clause Use the DISABLE clause to make the materialized view unavailable for use by the database server. When you disable a materialized view, the database server drops the data and all indexes for the view. The indexes must be reconstructed, and the view refreshed, after you re-enable the view.

{ ENABLE | DISABLE } USE IN OPTIMIZATION clause Use this clause to specify whether you want the materialized view to be available for use by the optimizer. If you specify DISABLE USE IN OPTIMIZATION, the materialized view is used only when executing queries that explicitly reference the view. The default is ENABLE USE IN OPTIMIZATION. See "Enabling and disabling optimizer use of a materialized view" [*SQL Anywhere Server - SQL Usage*].

ADD PCTFREE clause Specify the percentage of free space you want to reserve on each page. The free space is used if rows increase in size when the data is updated. If there is no free space on a page, every increase in the size of a row on that page requires the row to be split across multiple pages, causing row fragmentation and possible performance degradation.

The value of *percent-free-space* is an integer between 0 and 100. The value of 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself. If PCTFREE is not set, or is dropped, the default PCTFREE value is applied according to the database page size (200 bytes for a 4 KB page size, and 100 bytes for a 2 KB page size).

DROP PCTFREE clause Removes the PCTFREE setting currently in effect for the materialized view, and applies the default PCTFREE according to the database page size.

[NOT] ENCRYPTED clause Specify whether to encrypt the materialized view data. By default, materialized view data is not encrypted at creation time. To encrypt a materialized view, specify ENCRYPTED. To decrypt a materialized view, specify NOT ENCRYPTED.

Remarks

When you disable a materialized view, all indexes for it are dropped and must be recreated, if necessary, when the view is re-enabled.

After enabling a materialized view (ENABLE clause), you must execute a REFRESH MATERIALIZED VIEW statement to populate it with data.

After you disable a materialized view (DISABLE clause), it is no longer available for use by the database server for answering queries. Any views dependent on the materialized view are also disabled. The data in the materialized view is discarded; however, the definition for the view remains in the database. The DISABLE clause requires exclusive access not only to the view being disabled, but to any dependent views, since they are disabled too. See "Enabling and disabling materialized views" [*SQL Anywhere Server - SQL Usage*].

Table encryption must already be enabled on the database in order to encrypt a materialized view (ENCRYPT clause). The materialized view is then encrypted using the encryption key and algorithm specified at database creation time. See "Encrypting and decrypting materialized views" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be owner of the materialized view or have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE MATERIALIZED VIEW statement" on page 411
- "REFRESH MATERIALIZED VIEW statement" on page 621
- "DROP statement" on page 498
- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]
- "CREATE VIEW statement" on page 471
- "ALTER VIEW statement" on page 342

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example encrypts the EmployeeSalary materialized view:

```
ALTER MATERIALIZED VIEW EmployeeSalary
ENCRYPTED;
```


ALTER PROCEDURE statement

Use this statement to modify a procedure, or to enable and disable a procedure for replication with Sybase Replication Server. You must include the entire new procedure in the ALTER PROCEDURE statement.

You can use PROC as a synonym for PROCEDURE.

Syntax 1

ALTER PROCEDURE [*owner***.**]*procedure-name procedure-definition*

procedure-definition : CREATE PROCEDURE syntax

Syntax 2

ALTER PROCEDURE [*owner***.**]*procedure-name*
REPLICATE { **ON** | **OFF** }

Syntax 3

ALTER PROCEDURE [*owner***.**]*procedure-name* **SET HIDDEN**

Remarks

Syntax 1 The ALTER PROCEDURE statement is identical in syntax to the CREATE PROCEDURE statement except for the first word. Either version of the CREATE PROCEDURE statement can be altered.

Existing permissions on the procedure are maintained, and do not have to be reassigned. If a DROP PROCEDURE and CREATE PROCEDURE were carried out, execute permissions would have to be reassigned.

Syntax 2 If a procedure is to be replicated to other sites using Sybase Replication Server, you must set REPLICATE ON for the procedure.

Syntax 3 You can use SET HIDDEN to scramble the definition of the associated procedure and cause it to become unreadable. The procedure can be unloaded and reloaded into other databases.

This setting is irreversible. If you will need the original source again, you must maintain it outside the database.

If SET HIDDEN is used, debugging using the debugger will not show the procedure definition, nor will it be available through procedure profiling.

You cannot combine Syntax 2 with Syntax 1. You cannot combine Syntax 3 with either Syntax 1 or 2.

Permissions

Must be the owner of the procedure or have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE PROCEDURE statement" on page 414
- "ALTER FUNCTION statement" on page 310
- "DROP statement" on page 498
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to alter a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, a publication identifies replicated data in both consolidated and remote databases.

Syntax

ALTER PUBLICATION [*owner*.]*publication-name alterpub-clause*, …

alterpub-clause:
ADD *article-definition*
| **ALTER** *article-definition*
| { **DELETE** | **DROP** } **TABLE** [*owner*.]*table-name*
| **RENAME** *publication-name*

article-definition :
TABLE *table-name* [**(** *column-name*, … **)**]
[**WHERE** *search-condition*]
[**SUBSCRIBE BY** *expression*]
[**USING** ([**PROCEDURE**] [*owner***.**][*procedure-name*]
FOR UPLOAD { **INSERT** | **DELETE** | **UPDATE** }, ... **)**]

Remarks

This statement is applicable only to MobiLink and SQL Remote.

The ALTER PUBLICATION statement alters a publication in the database. The contribution to a publication from one table is called an article. Changes can be made to a publication by adding, modifying, or deleting articles, or by renaming the publication. If an article is modified, the entire definition of the modified article must be entered.

It is recommended that you perform a successful synchronization of the publication immediately before you alter it.

You cannot use the WHERE clause for publications that are defined as FOR DOWNLOAD ONLY or WITH SCRIPTED UPLOAD.

The SUBSCRIBE BY clause applies to SQL Remote only.

The USING clause is for scripted upload only.

You set options for a MobiLink publication with the ADD OPTION clause in the ALTER SYNCHRONIZATION SUBSCRIPTION statement or CREATE SYNCHRONIZATION SUBSCRIPTION statement.

Permissions

Must have DBA authority, or be the owner of the publication. Requires exclusive access to all tables referred to in the statement.

Side effects

Automatic commit.

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 427
- "DROP PUBLICATION statement [MobiLink] [SQL Remote]" on page 503
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*MobiLink - Client Administration*]
- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 328
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 445
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement adds the Customers table to the pub_contact publication.

```
ALTER PUBLICATION pub_contact
   ADD TABLE Customers;
```

ALTER REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to change the publisher's message system, or the publisher's address for a given message system, for a message type that has been created.

Syntax

ALTER REMOTE MESSAGE TYPE *message-system*
ADDRESS *address*

message-system: **FILE** | **FTP** | **MAPI** | **SMTP** | **VIM**

address: *string*

Parameters

message-system One of the message systems supported by SQL Remote. It must be one of the following values:

address A string containing a valid address for the specified message system.

Remarks

The statement changes the publisher's address for a given message type.

The Message Agent sends outgoing messages from a database by one of the supported message links. The extraction utility uses this address when executing the GRANT CONSOLIDATE statement in the remote database.

The address is the publisher's address under the specified message system. If it is an email system, the address string must be a valid email address. If it is a file-sharing system, the address string is a subdirectory of the directory specified by the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

Note
Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE REMOTE MESSAGE TYPE statement [SQL Remote]" on page 431

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement changes the publisher's address for the FILE message link to new_addr.

```
ALTER REMOTE MESSAGE TYPE file
ADDRESS 'new_addr';
```


ALTER SERVER statement

Use this statement to modify the attributes of a remote server.

Syntax

ALTER SERVER *server-name*
[**CLASS** *server-class*]
[**USING** *connection-info*]
[**CAPABILITY** *cap-name* { **ON** | **OFF** }]
[**CONNECTION CLOSE** [**CURRENT** | **ALL** | *connection-id*]]

server-class :
SAJDBC | **ASEJDBC** | **SAODBC** | **ASEODBC**
| **DB2ODBC** | **MSSODBC** | **ORAODBC** | **ODBC**

connection-info :
computer-name:*port-number*[/*dbname*] | *data-source-name*

Parameters

CLASS clause The CLASS clause is specified to change the server class.

For more information on server classes and how to configure a server, see "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*].

USING clause The USING clause is specified to change the server connection information. For information about *connection-info*, see "CREATE SERVER statement" on page 435.

CAPABILITY clause The CAPABILITY clause turns a server capability ON or OFF. Server capabilities are stored in the ISYSCAPABILITY system table. The names of these capabilities are stored in the ISYSCAPABILITYNAME system table. The ISYSCAPABILITYNAME system table contains no entries for a remote server until the first connection is made to that server. At the first connection, SQL Anywhere interrogates the database server about its capabilities and then populates the ISYSCAPABILITY table. For subsequent connections, the database server's capabilities are obtained from this table.

In general, you do not need to alter a server's capabilities. It may be necessary to alter capabilities of a generic server of class ODBC.

CONNECTION CLOSE clause
When a user creates a connection to a remote server, the remote connection is not closed until the user disconnects from the local database. The CONNECTION CLOSE clause allows you to explicitly close connections to a remote server. You may find this useful when a remote connection becomes inactive or is no longer needed.

The following SQL statements are equivalent and close the current connection to the remote server:

```
ALTER SERVER server-name CONNECTION CLOSE;

ALTER SERVER server-name CONNECTION CLOSE CURRENT;
```

You can close both ODBC and JDBC connections to a remote server using this syntax. You do not need DBA authority to execute either of these statements.

You can also disconnect a specific remote ODBC connection by specifying a connection ID, or disconnect all remote ODBC connections by specifying the ALL keyword. If you attempt to close a JDBC connection by specifying the connection ID or the ALL keyword, an error occurs. When the connection identified by *connection-id* is not the current local connection, the user must have DBA authority to be able to close the connection.

Remarks

The ALTER SERVER statement modifies the attributes of a server. These changes do not take effect until the next connection to the remote server.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "ISYSCAPABILITY system table" on page 727, and "ISYSCAPABILITYNAME system table" on page 727
- "CREATE SERVER statement" on page 435, and "DROP SERVER statement" on page 505
- "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*]
- "Troubleshooting remote data access" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example changes the server class of the Adaptive Server Enterprise server named ase_prod so its connection to SQL Anywhere is ODBC-based. Its Data Source Name is ase_prod.

```
ALTER SERVER ase_prod
CLASS 'ASEODBC'
USING 'ase_prod';
```

The following example changes a capability of server infodc.

```
ALTER SERVER infodc
CAPABILITY 'insert select' OFF;
```

The following example closes all connections to the remote server named rem_test.

```
ALTER SERVER rem_test
CONNECTION CLOSE ALL;
```

The following example closes the connection to the remote server named rem_test that has the connection ID 142536.

```
ALTER SERVER rem_test
CONNECTION CLOSE 142536;
```


ALTER SERVICE statement

Use this statement to alter a web service.

Syntax 1 - DISH service

ALTER SERVICE *service-name*
[**TYPE 'DISH'**]
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2 - SOAP service

ALTER SERVICE *service-name*
[**TYPE 'SOAP'**]
[**DATATYPE** { **ON** | **OFF** | **IN** | **OUT** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
[**AS** *statement*]

Syntax 3 - Miscellaneous services

ALTER SERVICE *service-name*
[**TYPE** { **'RAW'** | **'HTML'** | **'XML'** }]
[**URL** [**PATH**] { **ON** | **OFF** | **ELEMENTS** }]
[*common-attributes*]
[**AS** { *statement* | **NULL** }]

common-attributes:
[**AUTHORIZATION** { **ON** | **OFF** }]
[**SECURE** { **ON** | **OFF** }]
[**USER** { *user-name* | **NULL** }]

Parameters

service-name Identifies the service being altered.

service-type-string Identifies the type of the service. The type must be one of the listed service types. There is no default value.

AUTHORIZATION clause Determines whether users must specify a user name and password through HTTP basic authorization when connecting to the service. If authorization is OFF, the AS clause is required and a single user must be identified by the USER clause. All requests are run using that user's account and permissions.

If authorization is ON, all users must provide a user name and password. Optionally, you can limit the users that are permitted to use the service by providing a user or group name using the USER clause. If the user name is NULL, all known users can access the service.

The default value is ON. It is recommended that production systems be run with authorization turned on and that you grant permission to use the service by adding users to a group.

SECURE clause Indicates whether unsecured connections are accepted. ON indicates that only HTTPS connections are to be accepted. Service requests received on the HTTP port are automatically redirected to the HTTPS port. If set to OFF, both HTTP and HTTPS connections are accepted. The default value is OFF.

USER clause If authorization is disabled, this parameter becomes mandatory and specifies the user ID used to execute all service requests. If authorization is enabled (the default), this optional clause identifies the user or group permitted to access the service. The default value is NULL, which grants access to all users.

URL clause Determines whether URI paths are accepted and, if so, how they are processed. OFF indicates that nothing must follow the service name in a URI request. ON indicates that the remainder of the URI is interpreted as the value of a variable named url. ELEMENTS indicates that the remainder of the URI path is to be split at the slash characters into a list of up to 10 elements. The values are assigned to variables named url plus a numeric suffix of between 1 and 10; for example, the first three variable names are url1, url2, and url3. If fewer than 10 values are supplied, the remaining variables are set to NULL. If the service name ends with the forward slash character /, then url must be set to OFF. The default value is OFF.

GROUP clause Applies to DISH services only. Specifies a common prefix that controls which SOAP services the DISH service exposes. For example, specifying GROUP *xyz* exposes only SOAP services *xyz*/*aaaa*, *xyz*/*bbbb*, or *xyz*/*cccc,* but does not expose *abc/aaaa* or *xyzaaaa*. If no group name is specified, the DISH service exposes all the SOAP services in the database. SOAP services may be exposed by more than one DISH service. The same characters are permitted in group names as in service names.

DATATYPE clause Applies to SOAP services only. Controls whether data typing is supported for parameter inputs and/or result set outputs (responses) for all SOAP service formats. When supported, data typing allows a SOAP toolkit to parse and cast the data to the appropriate type. Parameter data types are exposed in the schema section of the Web Service Definition Language (WSDL) generated by the DISH service. Output data types are represented as XML schema type attributes for each column of data.

The following values are permitted for the DATATYPE clause:

- **ON** Support data typing for input parameters and result set responses.
- **OFF** Do not support data typing of input parameters and result set responses (the default).
- **IN** Support data typing of input parameters only.
- **OUT** Support data typing of result set responses only.

FORMAT clause Applies to DISH and SOAP services only. Generates output formats compatible with various types of SOAP clients, such as .NET or Java JAX-RPC. If the format of a SOAP service is not specified, the format is inherited from the service's DISH service declaration. If the DISH service also does not declare a format, it defaults to DNET, which is compatible with .NET clients. A SOAP service that does not declare a format may be used with different types of SOAP clients by defining multiple DISH services, each having a different FORMAT type.

statement If the statement is NULL, the URI must specify the statement to be executed. Otherwise, the specified SQL statement is the only one that can be executed through the service. SOAP services must have statements; DISH services must have none. The default value is NULL.

It is strongly recommended that all services run in production systems define a statement. The statement can be NULL only if authorization is enabled.

Format types

- **DNET** Microsoft DataSet format for use with .NET SOAP clients. DNET is the default FORMAT value and was the only format available before version 9.0.2.

- **CONCRETE** A platform-neutral DataSet format for use with clients such as JAX-RPC, or with clients that automatically generate interfaces based on the format of the returned data structure. Specifying this format type exposes an SimpleDataset element within the WSDL. This element describes the result set as a containment hierarchy of a rowset composed of an array of rows, each of which contains an array of column elements.
- **XML** A simple XML string format. The DataSet is returned as a string that can be passed to an XML parser. This format is the most portable between SOAP clients.

Service types

- **RAW** The result set is sent to the client without any further formatting. You can produce formatted documents by generating the required tags explicitly within your procedure.
- **HTML** The result set of a statement or procedure is automatically formatted into an HTML document that contains a table.
- **XML** The result set is returned as XML. If the result set is already XML, no additional formatting is applied. If it is not already XML, it is automatically formatted as XML. The effect is similar to that of using the FOR XML RAW clause in a SELECT statement.
- **SOAP** The result set is returned as a SOAP response. The format of the data is determined by the FORMAT clause. A request to a SOAP service must be a valid SOAP request, not just a simple HTTP request. For more information about the SOAP standards, see www.w3.org/TR/SOAP.
- **DISH** A DISH service (Determine SOAP Handler) acts as a proxy for those SOAP services identified by the GROUP clause, and generates a WSDL (Web Services Description Language) file for each of these SOAP services.

Remarks

The ALTER SERVICE statement makes changes to the ISYSWEBSERVICE system table, and allows the database server to act as a web server.

Permissions

Must have DBA authority.

Side effects

None.

See also

- “Using SOAP services” [*SQL Anywhere Server - Programming*]
- “CREATE SERVICE statement” on page 438
- “DROP SERVICE statement” on page 506
- “SYSWEBSERVICE system view” on page 807
- “-xs server option” [*SQL Anywhere Server - Database Administration*]
- “SQL Anywhere Web Services” [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

To set up a web server quickly, start a database server with the -xs (http or https) option, then execute the following statements:

```
CREATE SERVICE tables TYPE 'HTML';

ALTER SERVICE tables
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTAB;
```

After executing these statements, use any web browser to open the URL *http://localhost/tables*.

ALTER STATISTICS statement

Use this statement to control whether statistics are automatically updated on a column, or columns, in a table.

Syntax

ALTER STATISTICS
[**ON**] *table* [**(** *column1* [**,** *column2* ...] **)**]
AUTO UPDATE { **ENABLE** | **DISABLE** }

Parameters

ON The word ON is optional. Including it has no impact on the execution of the statement.

AUTO UPDATE clause Specify whether to enable or disable automatic updating of statistics for the column(s).

Remarks

During normal execution of queries, DML statements, and LOAD TABLE statements, the database server automatically maintains column statistics for use by the optimizer. The benefit of maintaining statistics for some columns may not outweigh the overhead necessary to generate them. For example, if a column is not queried often, or if it is subject to periodic mass changes that are eventually rolled back, there is little value in continually updating its statistics. Use the ALTER STATISTICS statement to suppress the automatic updating of statistics for these types of columns.

When automatic updating is disabled, you can still update the statistics for the column using the CREATE STATISTICS and DROP STATISTICS statements. However, you should only update them if it has been determined that it would have a positive impact on performance. Normally, column statistics should not be disabled.

Permissions

Must have DBA authority.

Side effects

If automatic updating has been disabled, the statistics may become out of date. Re-enabling will not immediately bring them up to date. Run the CREATE STATISTICS statement to recreate them, if necessary.

See also

- “CREATE STATISTICS statement” on page 442
- “DROP STATISTICS statement” on page 508

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example disables the automatic updating of statistics on the Street column in the Customers table:

```
ALTER STATISTICS Customers ( Street ) AUTO UPDATE DISABLE;
```


ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement in a SQL Anywhere remote database to alter the properties of a subscription of a MobiLink user to a publication.

Syntax

ALTER SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, …]
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option*=*value*, …]
[**ALTER OPTION** *option*=*value*, …]
[**DELETE** { **ALL OPTION** | **OPTION** *option*, … }]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** |**tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

TO clause Specify the name of a publication.

FOR clause Specify one or more MobiLink user names.

Omit the FOR clause to set the protocol type, protocol options, and extended options for a publication.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

TYPE clause This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

ADDRESS clause This clause specifies network protocol options, including the location of the MobiLink server.

For a complete list of protocol options, see "MobiLink client network protocol options" [*MobiLink - Client Administration*].

ADD OPTION, ALTER OPTION, DELETE OPTION, AND DELETE ALL OPTION clauses These clauses allow you to add, alter, delete, or delete all extended options. You may specify only one option in each clause.

The values for each option cannot contain the characters "**=**" or "**,**" or "**;**".

For a complete list of options, see "MobiLink SQL Anywhere Client Extended Options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 427
- "DROP PUBLICATION statement [MobiLink] [SQL Remote]" on page 503
- SQL Anywhere MobiLink clients: "Creating synchronization subscriptions" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example changes the address of the MobiLink server:

```
ALTER SYNCHRONIZATION SUBSCRIPTION
TO p1
FOR ml1
TYPE TCPIP
ADDRESS 'host=10.11.12.132;port=2439';
```


ALTER SYNCHRONIZATION USER statement [MobiLink]

Use this statement in a SQL Anywhere remote database to alter the properties of a MobiLink user.

Syntax

ALTER SYNCHRONIZATION USER *ml_username*
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option=value*, …]
[**ALTER OPTION** *option=value*, …]
[**DELETE** { **ALL OPTION** | **OPTION** *option* }]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** | **tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

TYPE clause This clause specifies the network protocol to use for synchronization.

For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

ADDRESS clause This clause specifies network protocol options, including the location of the MobiLink server.

For a complete list of protocol options, see "MobiLink client network protocol options" [*MobiLink - Client Administration*].

ADD OPTION, ALTER OPTION, DELETE OPTION, and DELETE ALL OPTION clauses These clauses allow you to add, modify, delete, or delete all extended options. You may specify only one option in each clause.

For a complete list of options, see "MobiLink SQL Anywhere Client Extended Options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 448
- "DROP SYNCHRONIZATION USER statement [MobiLink]" on page 511
- "MobiLink Users" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER TABLE statement

Use this statement to modify a table definition, disable dependent views, or enable a table to take part in Replication Server replication.

Syntax

ALTER TABLE [*owner.*]*table-name* { *alter-clause*, ... }

alter-clause :
ADD *create-clause*
| **ALTER** *column-name column-alteration*
| **ALTER** [**CONSTRAINT** *constraint-name*] **CHECK (** *condition* **)**
| **DROP** *drop-object*
| **RENAME** *rename-object*
| *table-alteration*

create-clause :
column-name [**AS**] *column-data-type* [*new-column-attribute ...*]
| *table-constraint*
| **PCTFREE** *integer*

column-alteration :
{ *column-data-type* | *alterable-column-attribute* } [*alterable-column-attribute ...*]
| **SET COMPUTE (** *compute-expression* **)**
| **ADD** [*constraint-name*] **CHECK (** *condition* **)**
| **DROP** { **DEFAULT** | **COMPUTE** | **CHECK** | **CONSTRAINT** *constraint-name* }

drop-object :
column-name
| **CHECK**
| **CONSTRAINT** *constraint-name*
| **UNIQUE** [**CLUSTERED**] **(** *index-columns-list* **)**
| **FOREIGN KEY** *fkey-name*
| **PRIMARY KEY**

rename-object :
new-table-name
| *column-name* **TO** *new-column-name*
| **CONSTRAINT** *constraint-name* **TO** *new-constraint-name*

table-alteration :
PCTFREE DEFAULT
| **REPLICATE** { **ON** | **OFF** }
| [**NOT**] **ENCRYPTED**

new-column-attribute :
NULL
| **DEFAULT** *default-value*
| **COMPRESSED**
| **INLINE** { *inline-length* | **USE DEFAULT** }
| **PREFIX** { *prefix-length* | **USE DEFAULT** }
| [**NO**] **INDEX**
| **IDENTITY**

| **COMPUTE (** *expression* **)**
| *column-constraint*

table-constraint :
[**CONSTRAINT** *constraint-name*] {
  CHECK (*condition* **)**
  | **UNIQUE** [**CLUSTERED** | **NONCLUSTERED**] **(** *column-name* [**ASC** | **DESC**], ... **)**
  | **PRIMARY KEY** [**CLUSTERED** | **NONCLUSTERED**] **(** *column-name* [**ASC** | **DESC**], ... **)**
  | *foreign-key*
 }

column-constraint :
[**CONSTRAINT** *constraint-name*] {
  CHECK (*condition* **)**
  | **UNIQUE** [**CLUSTERED** | **NONCLUSTERED**] [**ASC** | **DESC**]
  | **PRIMARY KEY** [**CLUSTERED** | **NONCLUSTERED**] [**ASC** | **DESC**]
  | **REFERENCES** *table-name* [**(** *column-name* **)**]
    [**MATCH** [**UNIQUE**] { **SIMPLE** | **FULL** }]
    [*actions*][**CLUSTERED** | **NONCLUSTERED**]
  | **NOT NULL**
 }

alterable-column-attribute :
 [**NOT**] **NULL**
| **DEFAULT** *default-value*
| [**CONSTRAINT** *constraint-name*] **CHECK** { **NULL** | **(** *condition* **)** }
| [**NOT**] **COMPRESSED**
| **INLINE** { *inline-length* | **USE DEFAULT** }
| **PREFIX** { *prefix-length* | **USE DEFAULT** }
| [**NO**] **INDEX**

default-value :
 special-value
| *string*
| *global variable*
| [**-**] *number*
| **(** *constant-expression* **)**
| *built-in-function* **(** *constant-expression* **)**
| **AUTOINCREMENT**
| **GLOBAL AUTOINCREMENT** [**(** *partition-size* **)**]
| **NULL**
| **TIMESTAMP**
| **UTC TIMESTAMP**
| **LAST USER**
| **USER**

special-value :
CURRENT {
  DATABASE
  | **DATE**
  | **REMOTE USER**
  | **TIME**
  | **TIMESTAMP**
  | **UTC TIMESTAMP**
  | **USER**
  | **PUBLISHER** }

foreign-key :
[**NOT NULL**] **FOREIGN KEY** [*role-name*]
  [**(** *column-name* [**ASC** | **DESC**], ... **)**
  REFERENCES *table-name*
  [**(** *pkey-column-list* **)**]
  [**MATCH** [**UNIQUE**] { **SIMPLE** | **FULL** }]
  [*actions*] [**CHECK ON COMMIT**] [**CLUSTERED**]
  [**FOR OLAP WORKLOAD**]

actions :
[**ON UPDATE** *action*] [**ON DELETE** *action*]

action :
CASCADE | **SET NULL** | **SET DEFAULT** | **RESTRICT**

Syntax 2 - Disabling view dependencies

ALTER TABLE [*owner.*]*table-name* {
 DISABLE VIEW DEPENDENCIES
}

Parameters

ADD column-name [AS] column-data-type [new-column-attribute ...] clause Use this syntax to add a new column to the table, specifying the data type and attributes for the column. For more information about what data type to specify, see "SQL Data Types" on page 47.

Possible column attributes include:

- **[NOT] NULL clause** Use this clause to specify whether to allow NULLs in the column. With the exception of bit type columns, new columns allow NULL values. Bit type columns automatically have the NOT NULL constraint applied when created.
- **DEFAULT clause** Sets the default value for the column. All rows in the column are populated with this value. For information about possible default values, see "CREATE TABLE statement" on page 450.
- **column-constraint clause**
  Use this clause to add a constraint to the column. With the exception of CHECK constraints, when a new constraint is added, the database server validates existing values to confirm that they satisfy the constraint. CHECK constraints are enforced only for operations that occur after the table alteration is complete. Possible column constraints include:
  - **CHECK clause** Use this subclause to add a check condition for the column.
  - **UNIQUE clause** Use this subclause to specify that values in the column must be unique, and whether to create a clustered or nonclustered index.
  - **PRIMARY KEY clause** Use this subclause to make the column a primary key, and whether to use a clustered index. For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].
  - **REFERENCES clause** Use this subclause to add or alter a reference to another table, to specify how matches are handled, and to specify whether to use a clustered index. For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

- **[NOT] NULL clause** Use this clause to specify whether to allow NULL values in the column. By default, NULLs are allowed.

- **COMPRESSED clause** Use this clause to compress the column.

- **INLINE and PREFIX clauses** When storing BLOBs (character and binary data types only), use the INLINE and PREFIX clauses to specify how much of a BLOB, in bytes, to keep within a row. For more information, see the INLINE and PREFIX clauses in "CREATE TABLE statement" on page 450.

- **[NO] INDEX** When storing BLOBs (character and binary data types only), use this clause to specify whether to build indexes for BLOB values. For more information, see the [NO] INDEX clause in "CREATE TABLE statement" on page 450.

> **Note**
> A BLOB index is not the same as a database index. A BLOB index is created to provide faster random access into BLOB data, whereas a database index is created to index values in one or more columns.

- **IDENTITY** This clause is equivalent to AUTOINCREMENT, and is provided for compatibility with T-SQL. See the description for AUTOINCREMENT in "CREATE TABLE statement" on page 450.

- **COMPUTE** Use this clause to ensure that the value in the column reflects the value of *expression*. For more information on what is allowed for the COMPUTE clause, see "CREATE TABLE statement" on page 450.

ADD table-constraint clause Use this clause to add a table constraint. Table constraints place limits on what columns in the table can hold. When adding or altering table constraints, the optional constraint name allows you to modify or drop individual constraints. Following is a list of the table constraints you can add.

- **CHECK** Use this subclause to add a check condition for the table. Table CHECK constraints fail when a value of FALSE is returned. A value of UNKNOWN allows a change to take place. For additional information on this constraint, see "CREATE TABLE statement" on page 450.

- **UNIQUE** Use this subclause to specify that values in the columns specified in *column-list* must be unique, and, optionally, whether to use a clustered index. For additional information on this constraint, see "CREATE TABLE statement" on page 450.

- **PRIMARY KEY** Use this subclause to add or alter the primary key for the table, and specify whether to use a clustered index. The table must not already have a primary key that was created by the CREATE TABLE statement or another ALTER TABLE statement. For additional information on this constraint, see "CREATE TABLE statement" on page 450.

  For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

- **foreign-key** Use this subclause to add a foreign key as a constraint. For additional information on this constraint, see "CREATE TABLE statement" on page 450.

ADD PCTFREE clause Specify the percentage of free space you want to reserve in each table page. The free space is used if rows increase in size when the data is updated. If there is no free space in a table page, every increase in the size of a row on that page requires the row to be split across multiple table pages,

causing row fragmentation and possible performance degradation. A free space percentage of 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high free space percentage causes each row to be inserted into a page by itself. If PCTFREE is not set, or is dropped, the default PCTFREE value is applied according to the database page size (200 bytes for a 4 KB, and up, page size). The value for PCTFREE is stored in the ISYSTAB system table. When PCTFREE is set, all subsequent inserts into table pages use the new value, but rows that were already inserted are not affected. The value persists until it is changed. The PCTFREE specification can be used for base, global temporary, or local temporary tables.

ALTER column-name column-alteration clause Use this clause to change attributes for the specified column. If a column is contained in a unique constraint, a foreign key, or a primary key, then the constraint or key must be deleted before the column can be modified. Following is a list of the alterations you can make. For further information about these attributes, see "CREATE TABLE statement" on page 450.

- **column-data-type clause** Use this clause to alter the length or data type of the column. If necessary, the data in the modified column is converted to the new data type. If a conversion error occurs, the operation will fail and the table is left unchanged. You cannot reduce the size of a column. For example, you cannot change a column from a VARCHAR(100) to a VARCHAR(50).
- **[NOT] NULL clause** Use this clause to change whether NULLs are allowed in the column. If NOT NULL is specified, and the column value is NULL in any of the existing rows, then the operation fails and the table is left unchanged.
- **CHECK NULL** Use this clause to delete all check constraints for the column.
- **DEFAULT clause** Use this clause to change the default value for the column.
- **DEFAULT NULL clause** Use this clause to remove the default value for the column.
- **[CONSTRAINT constraint-name] CHECK { NULL | (condition) } clause** Use this clause to add a CHECK constraint on the column.
- **[NOT] COMPRESSED clause** Use this clause to change whether the column is compressed.
- **INLINE and PREFIX clauses** Use the INLINE and PREFIX clauses with columns that contain BLOBs to specify how much of a BLOB, in bytes, to keep within a row. For more information about how to set the INLINE and PREFIX values, see the corresponding sections for the INLINE and PREFIX clauses in "CREATE TABLE statement" on page 450.
- **[NO] INDEX clause** Use this clause to specify whether to build indexes on large BLOBs in this column. For more information about how to use this clause, see the corresponding section for the [NO] INDEX clause in the "CREATE TABLE statement" on page 450.
- **SET COMPUTE clause** Use this clause to change the expression associated with the computed column. The values in the column are recalculated when the statement is executed, and the statement fails if the new expression is invalid. For more information on what is allowed for the COMPUTE expression, see "CREATE TABLE statement" on page 450.

ALTER CONSTRAINT constraint-name CHECK clause Use this clause to alter a named check constraint for the table.

DROP clause Use this clause to drop a column, a table constraint, or an index. Possible objects to drop for tables or columns include:

- **DROP DEFAULT** Drops the default value set for the table or specified column. Existing values do not change.
- **DROP COMPUTE** Removes the COMPUTE attribute for the specified column. This statement does not change any existing values in the table.
- **DROP CHECK** Drops all CHECK constraints for the table or specified column. DELETE CHECK is also accepted.
- **DROP CONSTRAINT constraint-name** Drops the named constraint for the table or specified column. DELETE CONSTRAINT is also accepted.
- **DROP column-name** Drops the specified column from the table. DELETE *column-name* is also accepted. If the column is contained in any index, unique constraint, foreign key, or primary key, then the index, constraint, or key must be deleted before the column can be deleted. This does not delete CHECK constraints that refer to the column.
- **DROP UNIQUE (column-name ...)** Drop the unique constraints on the specified column(s). Any foreign keys referencing this unique constraint are also deleted. DELETE UNIQUE (*column-name* ...) is also accepted.
- **DROP FOREIGN KEY fkey-name** Drop the specified foreign key. DELETE FOREIGN KEY *fkey-name* is also accepted.
- **DROP PRIMARY KEY** Drop the primary key. All foreign keys referencing the primary key for this table are also deleted. DELETE PRIMARY KEY is also accepted.

RENAME clause Use this clause to rename a table, column or constraint.

- **RENAME new-table-name** Change the name of the table to *new-table-name*. Note that any applications using the old table name must be modified, as necessary. After the renaming operation succeeds, foreign keys with ON UPDATE or ON DELETE actions must be dropped and re-created, as the system-created triggers used to implement these actions will continue to refer to the old name.
- **RENAME column-name TO new-column-name** Change the name of the column to the *new-column-name*. Note that any applications using the old column name will need to be modified, as necessary. After the renaming operation succeeds, foreign keys with ON UPDATE or ON DELETE actions must be dropped and re-created, as the system-created triggers used to implement these actions will continue to refer to the old name.
- **RENAME CONSTRAINT constraint-name TO new-constraint-name** Change the name of the constraint to the *new-constraint-name*.

table-alteration clause Use this clause to alter attributes for the table.

- **PCTFREE DEFAULT** Use this clause to change the percent free setting for the table to the default (200 bytes for a 4 KB, and up, page size).

- **REPLICATE { ON | OFF }** Use this clause to change whether the table is included during replication. When a table has REPLICATE ON, all changes to the table are sent to Replication Server for replication. The replication definitions in Replication Server are used to decide which table changes are sent to other sites.
- **[NOT] ENCRYPTED** Use this clause to change whether the table is encrypted. Table encryption must already be enabled on the database in order to encrypt a table. The table is encrypted using the encryption key and algorithm specified at database creation time. See "Enabling table encryption" [*SQL Anywhere Server - Database Administration*]. After encrypting a table, data for that table in temporary files and in the transaction log still exists in unencrypted form. Restart the database to remove the temporary files. Run the Backup utility (dbbackup) with the -o option, or use the BACKUP statement, to back up the transaction log and start a new one. See "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*] or "BACKUP statement" on page 346.

  Once table encryption is enabled, table pages for the encrypted table, associated index pages, and temporary file pages are encrypted, as well as the transaction log pages that contain transactions on encrypted tables.

DISABLE VIEW DEPENDENCIES clause Use this clause to disable any non-materialized views that are dependent on the table. To re-enable the views disabled by this clause, you must execute an ALTER VIEW ... ENABLE statement for each view. This clause does not disable dependent materialized views; you must disable them using the ALTER MATERIALIZED VIEW ... DISABLE statement. Consequently, this clause cannot be used on a table while it has valid dependent materialized views. See "ALTER MATERIALIZED VIEW statement" on page 313.

Remarks

The ALTER TABLE statement changes table attributes (column definitions, constraints, and so on) in an existing table.

The database server keeps track of object dependencies in the database. Alterations to the schema of a table may impact dependent views. Also, if there are materialized views that are dependent on the table you are attempting to alter, you must first disable them using the ALTER MATERIALIZED VIEW ... DISABLE statement. For information on view dependencies, see "View dependencies" [*SQL Anywhere Server - SQL Usage*].

You cannot use ALTER TABLE on a local temporary table.

ALTER TABLE is prevented whenever the statement affects a table that is currently being used by another connection. ALTER TABLE can be time-consuming, and the database server does not process other requests referencing the table while the statement is being processed.

For more information on using the CLUSTERED option, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be one of the following:

- The owner of the table.
- A user with DBA authority.

- A user who has been granted ALTER permission on the table.

ALTER TABLE requires exclusive access to the table.

Global temporary tables cannot be altered unless all users that have referenced the temporary table have disconnected.

Cannot be used within a snapshot transaction. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit.

A checkpoint is carried out at the beginning of the ALTER TABLE operation, and further checkpoints are suspended until the ALTER operation completes.

Once you alter a column or table, any stored procedures, views, or other items that refer to the altered column may no longer work.

If you change the declared length or type of a column, or drop a column, the statistics for that column are dropped. For information on how to generate new statistics, see "Updating column statistics" [*SQL Anywhere Server - SQL Usage*].

See also

- "CREATE TABLE statement" on page 450
- "DROP statement" on page 498
- "SQL Data Types" on page 47
- "Altering tables" [*SQL Anywhere Server - SQL Usage*]
- "Special values" on page 30
- "Using table and column constraints" [*SQL Anywhere Server - SQL Usage*]
- "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "Enabling table encryption" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** ADD COLUMN is a core feature. Other clauses are vendor extensions or implementation of specific, named extensions to SQL/2003.

Example

The following example adds a new column to the Employees table showing which office they work in.

```
ALTER TABLE Employees
ADD Office CHAR(20) DEFAULT 'Boston';
```

The following example drops the Office column from the Employees table.

```
ALTER TABLE Employees
DROP Office;
```

The Street column in the Customers table can currently hold up to 35 characters. To allow it to hold up to 50 characters, execute the following:

```
ALTER TABLE Customers
ALTER Street CHAR(50);
```


The following example adds a column to the Customers table, assigning each customer a sales contact.

```
ALTER TABLE Customers
ADD SalesContact INTEGER
REFERENCES Employees ( EmployeeID )
ON UPDATE CASCADE
ON DELETE SET NULL;
```

This foreign key is constructed with cascading updates and is set to NULL on deletes. If an employee has their employee ID changed, the column is updated to reflect this change. If an employee leaves the company and has their employee ID deleted, the column is set to NULL.

ALTER TRIGGER statement

Use this statement to replace a trigger definition with a modified version.

You must include the entire new trigger definition in the ALTER TRIGGER statement.

Syntax 1

ALTER TRIGGER *trigger-name trigger-definition*

trigger-definition : CREATE TRIGGER syntax

Syntax 2

ALTER TRIGGER *trigger-name* **ON** [*owner.*] *table-name* **SET HIDDEN**

Remarks

Syntax 1 The ALTER TRIGGER statement is identical in syntax to the CREATE TRIGGER statement except for the first word. For information on *trigger-definition*, see "CREATE TRIGGER statement" on page 462 and "CREATE TRIGGER statement [T-SQL]" on page 468.

Either the Transact-SQL or Watcom-SQL form of the CREATE TRIGGER syntax can be used.

Syntax 2 You can use SET HIDDEN to scramble the definition of the associated trigger and cause it to become unreadable. The trigger can be unloaded and reloaded into other databases. If SET HIDDEN is used, debugging using the debugger will not show the trigger definition, nor will it be available through procedure profiling.

Note
The SET HIDDEN operation is irreversible.

Permissions

Must be the owner of the table on which the trigger is defined, or be user DBA, or have ALTER permissions on the table and have RESOURCE authority.

Side effects

Automatic commit.

See also

- "CREATE TRIGGER statement" on page 462
- "CREATE TRIGGER statement [T-SQL]" on page 468
- "DROP statement" on page 498
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER VIEW statement

Use this statement to replace a view definition with a modified version.

Syntax 1

ALTER VIEW
[*owner*.]*view-name* [**(** *column-name*, ... **)**] **AS** *select-statement*
[**WITH CHECK OPTION**]

Syntax 2

ALTER VIEW
[*owner*.]*view-name* { **SET HIDDEN** | **RECOMPILE** | **DISABLE** | **ENABLE** }

Parameters

AS clause The purpose and syntax of this clause is identical to that of the CREATE VIEW statement. See "CREATE VIEW statement" on page 471.

WITH CHECK OPTION clause The purpose and syntax of this clause is identical to that of the CREATE VIEW statement. See "CREATE VIEW statement" on page 471.

SET HIDDEN clause Use the SET HIDDEN clause to obfuscate the definition of the view and cause the view to become hidden from view, for example in Sybase Central. Explicit references to the view will still work, however.

> **Note**
> The SET HIDDEN operation is irreversible.

RECOMPILE clause Use the RECOMPILE clause to re-create the column definitions for the view. This clause is identical in functionality to the ENABLE clause, except that you may decide to use it on a view that is not disabled.

DISABLE clause Use the DISABLE clause to disable the view from use by the database server.

ENABLE clause Use the ENABLE clause to enable a disabled view. Enabling the view causes the database server to re-create the column definitions for the view. Before you enable a view, you must enable any views upon which it depends.

Remarks

When you alter a view, existing permissions on the view are maintained, and do not have to be reassigned. Instead of using the ALTER VIEW statement, you could also drop the view and recreate it using the DROP VIEW and CREATE VIEW, respectively. However, if you do so, permissions on the view need to be reassigned.

After completing the view alteration using Syntax 1, the database server recompiles the view. Depending on the type of change you made, if there are dependent views, the database server will attempt to recompile them as well. If you have made a change that impacts a dependent view, you may need to alter the definition for the dependent view as well. For more information on view alterations and how they impact view dependencies, see "View dependencies" [*SQL Anywhere Server - SQL Usage*].

Caution
If the SELECT statement defining the view contained an asterisk (*), the number of the columns in the view may change if columns have been added or deleted from the underlying tables. The names and data types of the view columns may also change.

Syntax 1 This syntax is used to alter the structure of the view. Unlike altering tables where your change may be limited to individual columns, altering the structure of a view requires you to replace the entire view definition with a new definition, much as you would for creating the view. For a description of the parameters used to define the structure of a view, see "CREATE VIEW statement" on page 471.

Syntax 2 This syntax is used to change attributes for the view, such as whether the view definition is hidden.

When you use SET HIDDEN, the view can be unloaded and reloaded into other databases. If SET HIDDEN is used, debugging using the debugger will not show the view definition, nor will it be available through procedure profiling. If you need to change the definition of a hidden view, you must drop the view and create it again using the CREATE VIEW statement.

When you use the DISABLE clause, the view is no longer available for use by the database server for answering queries. Disabling a view is similar to dropping it, except that the view definition remains in the database. Disabling a view also disables any dependent views. Therefore, the DISABLE clause requires exclusive access not only to the view being disabled, but also any dependent views, since they are disabled too.

Permissions

Must be owner of the view or have DBA authority.

Side effects

Automatic commit.

All procedures and triggers are unloaded from memory, so that any procedure or trigger that references the view reflects the new view definition. The unloading and loading of procedures and triggers can have a performance impact if you are regularly altering views.

See also

- "CREATE VIEW statement" on page 471
- "DROP statement" on page 498
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]
- "CREATE MATERIALIZED VIEW statement" on page 411
- "ALTER MATERIALIZED VIEW statement" on page 313

Standards and compatibility

- **SQL/2003** Vendor extension.

ATTACH TRACING statement

Use this statement to start a diagnostic tracing session. That is, to start sending diagnostic information to the diagnostic tables.

Syntax

ATTACH TRACING TO { **LOCAL DATABASE** | *connect-string* }
[**LIMIT** { *size* | *history* }]

connect-string : the connection string for the database

size : **SIZE** *nnn* { **MB** | **GB** }

history : **HISTORY** *nnn* { **MINUTES** | **HOURS** | **DAYS** }

nnn : *integer*

Parameters

connstr The connection string required to connect to database receiving the tracing information. This parameter is only required when the database being profiled is different from the database receiving the data.

limit The volume limit of data stored in the tracing database, either by size, or by length of time.

Remarks

The ATTACH TRACING statement is primarily used by the Diagnostic Tracing wizard in Sybase Central. However, you can also run it manually. You must run if from the database you want to profile.

The ATTACH TRACING statement is used to start a tracing session for the database you want to profile. You can only use it once a tracing level has been set. You can set the tracing level using Sybase Central, or using the sa_set_tracing_level system procedure.

Once a session is started, tracing information is generated according to the tracing levels set in the sa_diagnostic_tracing_level table. You can send the tracing data to *tracing tables* within the same database that is being profiled, by specifying LOCAL DATABASE. Alternatively, you can send the tracing data to a separate *tracing database* by specifying a connection string (*connect-string*) to that database. The tracing database must already exist, and you must have permissions to access it.

You can limit the amount of tracing data to store using the LIMIT SIZE or LIMIT HISTORY clauses. Use the LIMIT SIZE clause when you want to limit the volume of tracing data to a certain size, as measured in megabytes or gigabytes. Use the LIMIT HISTORY clause to limit the volume of tracing data to a period of time, as measured in minutes, hours, or days. For example, `HISTORY 8 DAYS` limits the amount of tracing data stored in the tracing database to 8 days' worth.

In order to start a tracing session, TCP/IP must be running on the database server(s) on which the tracing database and production database are running. See "Using the TCP/IP protocol" [*SQL Anywhere Server - Database Administration*].

Packets that contain potentially sensitive data are visible on the network interface, even when tracing to a local database. For security purposes, you can specify encryption in the connection string.

To see the current tracing levels set for a database, look in the sa_diagnostic_tracing_level table. See "sa_diagnostic_tracing_level table" on page 748.

To see where tracing data is being sent to, examine the SendingTracingTo database property. See "Database-level properties" [*SQL Anywhere Server - Database Administration*].

Permissions

Must be connected to the database being profiled and must have DBA authority.

Side effects

None.

See also

- "DETACH TRACING statement" on page 496
- "REFRESH TRACING LEVEL statement" on page 623
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]
- "sa_set_tracing_level system procedure" on page 925

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example sets the tracing level to 1 using the sa_set_tracing_level system procedure. Then it starts a tracing session. Tracing data generated for the local database will be sent to the mytracingdb tracing database on another computer, as shown by the specified connection string. A maximum of two hours of tracing data will be maintained during the tracing session. Note that the ATTACH TRACING statement example is all on one line.

```
CALL sa_set_tracing_level( 1 );
ATTACH TRACING TO
'uid=DBA;pwd=sql;eng=remotedbsrv10;dbn=mytracingdb;links=tcpip'
 LIMIT HISTORY 2 HOURS;
```


BACKUP statement

Use this statement to back up a database and transaction log.

Syntax 1 (image backup)

BACKUP DATABASE
DIRECTORY *backup-directory*
[**WAIT BEFORE START**]
[**WAIT AFTER END**]
[**DBFILE ONLY**]
[**TRANSACTION LOG ONLY**]
[**TRANSACTION LOG RENAME** [**MATCH**]]
[**TRANSACTION LOG TRUNCATE**]
[**ON EXISTING ERROR**]
[**WITH COMMENT** *comment string*]
[**HISTORY** { **ON** | **OFF** }]
[**AUTO TUNE WRITERS** { **ON** | **OFF** }]
[**WITH CHECKPOINT LOG** { **AUTO** | **COPY** | **NO COPY** | **RECOVER** }]

backup-directory : { *string* | *variable* }

Syntax 2 (archive backup)

BACKUP DATABASE TO *archive-root*
[**WAIT BEFORE START**]
[**WAIT AFTER END**]
[**DBFILE ONLY**]
[**TRANSACTION LOG ONLY**]
[**TRANSACTION LOG RENAME** [**MATCH**]]
[**TRANSACTION LOG TRUNCATE**]
[**ATTENDED** { **ON** | **OFF** }]
[**WITH COMMENT** *comment string*]
[**HISTORY** { **ON** | **OFF** }]

archive-root : { *string* | *variable* }

comment-string : *string*

Parameters

backup-directory The target location on disk for the backup files, relative to the database server's current directory at startup. If the directory does not exist, it is created. Specifying an empty string as a directory allows you to rename or truncate the log without first making a copy of it.

WAIT BEFORE START clause Use this clause to ensure that the backup copy of the database does not contain any information required for recovery. In particular, it ensures that the rollback log for each connection is empty.

If a backup is carried out using this clause, you can start the backup copy of the database in read-only mode and validate it. By enabling validation of the backup database, you can avoid making an additional copy of the database.

WAIT AFTER END clause Use this clause if the transaction log is being renamed or truncated. It ensures that all transactions are completed before the log is renamed or truncated. If this clause is used, the backup must wait for other connections to commit or rollback any open transactions before finishing.

DBFILE ONLY clause Use this clause to cause backup copies of the main database file and any associated dbspaces to be made. The transaction log is not copied. You cannot use the DBFILE ONLY clause with the TRANSACTION LOG RENAME or TRANSACTION LOG TRUNCATE clauses.

TRANSACTION LOG ONLY clause Use this clause to cause a backup copy of the transaction log to be made. No other database files are copied.

TRANSACTION LOG RENAME [MATCH] clause This clause causes the database server to rename the current transaction log at the completion of the backup. If the MATCH keyword is omitted, the backup copy of the log will have the same name as the current transaction log for the database. If you supply the MATCH keyword, the backup copy of the transaction log is given a name of the form *YYMMDDnn.log*, to match the renamed copy of the current transaction log. Using the MATCH keyword enables the same statement to be executed several times without writing over old data.

TRANSACTION LOG TRUNCATE clause If this clause is used, the current transaction log is truncated and restarted at the completion of the backup.

archive-root clause The file name or tape drive device name for the archive file.

To back up to tape, you must specify the device name of the tape drive. For example, on NetWare the first tape drive is \\.*tape0*. The number automatically appended to the end of the archive file name is incremented each time you execute an archive backup.

The backslash (\ ) is an escape character in SQL strings, so each backslash must be doubled. For more information on escape characters and strings, see "Strings" on page 8.

ON EXISTING ERROR clause This clause applies only to image backups. By default, existing files are overwritten when you execute a BACKUP DATABASE statement. If this clause is used, an error occurs if any of the files to be created by the backup already exist.

ATTENDED clause The clause applies only when backing up to a tape device. ATTENDED ON (the default) indicates that someone is available to monitor the status of the tape drive and to place a new tape in the drive when needed. A message is sent to the application that issued the BACKUP DATABASE statement if the tape drive requires intervention. The database server then waits for the drive to become ready. This may happen, for example, when a new tape is required.

If ATTENDED OFF is specified and a new tape is required or the drive is not ready, no message is sent and an error is given.

WITH COMMENT clause This clause records a comment in the backup history file. For archive backups, the comment is also recorded in the archive file.

HISTORY clause By default, each backup operation appends a line to the *backup.syb* file. You can prevent updates to the *backup.syb* file by specifying HISTORY OFF. You may want to prevent the file from being updated if any of the following conditions apply:

- your backups occur frequently
- there is no procedure to periodically archive or delete the *backup.syb* file

- disk space is very limited

AUTO TUNE WRITERS clause When the backup starts, one thread is dedicated to writing the backup files to the backup directory. However, if the backup directory is on a device that can handle an increased writer load (such as a RAID array), then overall backup performance can be improved by increasing the number of threads acting as writers. If this clause is ON, the database server periodically examines the read and write performance from all of the devices taking part in the backup. If the overall backup speed can be improved by creating another writer, then the database server creates another writer. This option is ON by default.

WITH CHECKPOINT LOG clause This clause specifies how the backup processes the database files before writing them to the destination directory. The choice of whether to apply pre-images during a backup, or copy the checkpoint log as part of the backup, has performance implications. The default setting is AUTO.

- **COPY** This option cannot be used with the WAIT BEFORE START clause of the BACKUP statement.

  When you specify COPY, the backup reads the database files without applying any modified pages. The checkpoint log in its entirety, as well as the system dbspace, is copied to the backup directory. The next time the database is started, the database server automatically recovers the database to the state it was in as of the checkpoint at the time the backup started.

  Because pages do not have to be written to the temporary file, using this option can provide better backup performance, and reduce internal server contention for other connections that are operating during a backup. However, since the checkpoint log contains original images of modified pages, it will grow in the presence of database updates. With copy specified, the backed-up copy of the database files may be larger than the database files at the time the backup started. The COPY option should be used if disk space in the destination directory is not an issue.

- **NO COPY** When you specify NO COPY, the checkpoint log is not copied as part of the backup. This option causes modified pages to be saved in the temporary file so that they can be applied to the backup as it progresses. The backup copies of the database files will be the same size as the database when the backup operation commenced.

  This option results in smaller backed up database files, but the backup may proceed more slowly, and possibly decrease performance of other operations in the database server. It is useful in situations where space on the destination drive is limited.

- **RECOVER** When you specify RECOVER, the database server copies the checkpoint log (as with the COPY option), but applies the checkpoint log to the database when the backup is complete. This restores the backed up database files to the same state (and size) that they were in at the start of the backup operation. This option is useful if space on the backup drive is limited (it requires the same amount of space as the COPY option for backing up the checkpoint log, but the resulting file size is smaller).

- **AUTO** When you specify AUTO, the database server checks the amount of available disk space on the volume hosting the backup directory. If there is at least twice as much disk space available as the size of the database at the start of the backup, then this option behaves as if copy were specified. Otherwise, it behaves as NO COPY. AUTO is the default behavior.

Remarks

The BACKUP statement performs a server-side backup. To perform a client-side backup, use the dbbackup utility. See "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*].

Each backup operation, whether image or archive, updates a history file called *backup.syb*. This file records the BACKUP and RESTORE operations that have been performed on a database server. For information about how the location of the *backup.syb* file is determined, see "SALOGDIR environment variable" [*SQL Anywhere Server - Database Administration*].

Syntax 1 (image backup) An image backup creates copies of each of the database files, in the same way as the Backup utility (dbbackup). By default, the Backup utility makes the backup on the client computer, but you can specify the -s option to create the backup on the database server when using the Backup utility. In the case of the BACKUP DATABASE statement, however, the backup can only be made on the database server.

Optionally, only the database file(s) or transaction log can be saved. The log may also be renamed or truncated after the backup has completed.

Alternatively, you can specify an empty string as a directory to rename or truncate the log without copying it first. This is particularly useful in a replication environment where space is a concern. You can use this feature with an event handler on transaction log size to rename the log when it reaches a given size, and with the delete_old_logs option to delete the log when it is no longer needed.

To restore from an image backup, copy the saved files back to their original locations and reapply transaction logs as described in "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*].

Syntax 2 (archive backup) An archive backup creates a single file holding all the required backup information. The destination can be either a file name or a tape drive device name.

There can be only one backup on a given tape. The tape is ejected at the end of the backup.

Only one archive per tape is allowed, but a single archive can span multiple tapes. To restore a database from an archive backup, use the RESTORE DATABASE statement.

If a RESTORE DATABASE statement references an archive file containing only a transaction log, the statement must specify a file name for the location of the restored database file, even if that file does not exist. For example, to restore from an archive containing only a log to the directory *C:\MYNEWDB*, the RESTORE DATABASE statement is:

```
RESTORE DATABASE 'c:\mynewdb\my.db' FROM archive-root
```

Permissions

Must have DBA, REMOTE DBA, or BACKUP authority.

Side effects

Causes a checkpoint.

See also

- "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*]
- "Making image backups" [*SQL Anywhere Server - Database Administration*]
- "RESTORE DATABASE statement" on page 631

- "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*]
- "EXECUTE IMMEDIATE statement [SP]" on page 519
- "Understanding parallel database backups" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.
- **Windows CE** Only the BACKUP DATABASE DIRECTORY syntax (syntax 1 above) is supported on Windows CE.

Example

Back up the current database and the transaction log, each to a different file, and rename the existing transaction log. An image backup is created.

```
BACKUP DATABASE
DIRECTORY 'd:\\temp\\backup'
TRANSACTION LOG RENAME;
```

The option to rename the transaction log is useful especially in replication environments, where the old transaction log is still required.

Back up the current database and transaction log to tape:

```
BACKUP DATABASE
TO '\\\\.\\tape0';
```

Rename the log without making a copy:

```
BACKUP DATABASE DIRECTORY ' '
TRANSACTION LOG ONLY
TRANSACTION LOG RENAME;
```

Execute the BACKUP DATABASE statement with a dynamically-constructed directory name:

```
CREATE EVENT NightlyBackup
SCHEDULE
START TIME '23:00' EVERY 24 HOURS
HANDLER
BEGIN
    DECLARE dest LONG VARCHAR;
    DECLARE day_name CHAR(20);

    SET day_name = DATENAME( WEEKDAY, CURRENT DATE );
    SET dest = 'd:\\backups\\' || day_name;
            BACKUP DATABASE DIRECTORY dest
    TRANSACTION LOG RENAME;
END;
```


BEGIN statement

Use this statement to group SQL statements together.

Syntax

[*statement-label* :]
BEGIN [[**NOT**] **ATOMIC**]
  [*local-declaration;* …]
  statement-list
  [**EXCEPTION** [*exception-case* …]]
END [*statement-label*]

local-declaration :
 variable-declaration
| *cursor-declaration*
| *exception-declaration*
| *temporary-table-declaration*

variable-declaration :
DECLARE *variable-name data-type*

exception-declaration :
DECLARE *exception-name* **EXCEPTION**
FOR SQLSTATE [**VALUE**] *string*

exception-case :
 WHEN *exception-name* [, …] **THEN** *statement-list*
| **WHEN OTHERS THEN** *statement-list*

Parameters

local-declaration Immediately following the BEGIN, a compound statement can have local declarations for objects that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.

statement-label If the ending *statement-label* is specified, it must match the beginning *statement-label*. The LEAVE statement can be used to resume execution at the first statement after the compound statement. The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

For a complete description of compound statements and exception handling, see "Errors and warnings in procedures and triggers" [*SQL Anywhere Server - SQL Usage*].

ATOMIC An atomic statement is a statement executed completely or not at all. For example, an UPDATE statement that updates thousands of rows might encounter an error after updating many rows. If the statement does not complete, all changes revert back to their original state. Similarly, if you specify that the BEGIN statement is atomic, the statement is executed either in its entirety or not at all.

Remarks

The body of a procedure or trigger is a compound statement. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword BEGIN and ends with the keyword END.

Permissions

None.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 483
- "CONTINUE statement [T-SQL]" on page 373
- "SIGNAL statement" on page 673
- "RESIGNAL statement" on page 630
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]
- "Atomic compound statements" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The body of a procedure or trigger is a compound statement.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION FOR
      SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
      SELECT CompanyName, CAST(
            sum( SalesOrderItems.Quantity *
            Products.UnitPrice ) AS INTEGER) VALUE
      FROM Customers
            LEFT OUTER JOIN SalesOrders
            LEFT OUTER JOIN SalesOrderItems
            LEFT OUTER JOIN Products
      GROUP BY CompanyName;
   DECLARE ThisValue INT;

   DECLARE ThisCompany CHAR( 35 );
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
         INTO ThisCompany, ThisValue;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop;
      END IF;
```



```
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue;
         SET TopCompany = ThisCompany;
      END IF;
   END LOOP CustomerLoop;
   CLOSE curThisCust;
END;
```


BEGIN TRANSACTION statement [T-SQL]

Use this statement to begin a user-defined transaction.

Syntax

BEGIN TRAN[**SACTION**] [*transaction-name*]

Remarks

The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. Use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the BEGIN TRANSACTION statement increases the nesting level of transactions by one. The nesting level is decreased by a COMMIT statement. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Both Adaptive Server Enterprise and SQL Anywhere have two transaction modes.

The default Adaptive Server Enterprise transaction mode, called unchained mode, commits each statement individually, unless an explicit BEGIN TRANSACTION statement is executed to start a transaction. In contrast, the ISO SQL/2003 compatible chained mode only commits a transaction when an explicit COMMIT is executed or when a statement that carries out an autocommit (such as data definition statements) is executed.

You can control the mode by setting the chained database option. The default setting for ODBC and embedded SQL connections in SQL Anywhere is On, in which case SQL Anywhere runs in chained mode. (ODBC users should also check the AutoCommit ODBC setting). The default for TDS connections is Off. See “chained option [compatibility]” [*SQL Anywhere Server - Database Administration*].

In unchained mode, a transaction is implicitly started before any data retrieval or modification statement. These statements include: DELETE, INSERT, OPEN, FETCH, SELECT, and UPDATE. You must still explicitly end the transaction with a COMMIT or ROLLBACK statement.

You cannot alter the chained option within a transaction.

Caution
When calling a stored procedure, you should ensure that it operates correctly under the required transaction mode.

The current nesting level is held in the global variable @@trancount. The @@trancount variable has a value of zero before the first BEGIN TRANSACTION statement is executed, and only a COMMIT executed when @@trancount is equal to one makes changes to the database permanent.

A ROLLBACK statement without a transaction or savepoint name always rolls back statements to the outermost BEGIN TRANSACTION (explicit or implicit) statement, and cancels the entire transaction.

Permissions

None.

Side effects

None.

See also

- "COMMIT statement" on page 367
- "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "ROLLBACK statement" on page 642
- "SAVEPOINT statement" on page 647
- "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following batch reports successive values of @@trancount as 0, 1, 2, 1, and 0. The values are printed on the Server Messages window.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT
PRINT @@trancount
COMMIT
PRINT @@trancount
```

You should not rely on the value of @@trancount for more than keeping track of the number of explicit BEGIN TRANSACTION statements that have been issued.

When Adaptive Server Enterprise starts a transaction implicitly, the @@trancount variable is set to 1. SQL Anywhere does not set the @@trancount value to 1 when a transaction is started implicitly. Consequently, the SQL Anywhere @@trancount variable has a value of zero before any BEGIN TRANSACTION statement (even though there is a current transaction), while in Adaptive Server Enterprise (in chained mode) it has a value of 1.

For transactions starting with a BEGIN TRANSACTION statement, @@trancount has a value of 1 in both SQL Anywhere and Adaptive Server Enterprise after the first BEGIN TRANSACTION statement. If a transaction is implicitly started with a different statement, and a BEGIN TRANSACTION statement is then executed, @@trancount has a value of 2 in both SQL Anywhere, and Adaptive Server Enterprise after the BEGIN TRANSACTION statement.

BREAK statement [T-SQL]

Use this statement to exit a compound statement or loop.

Syntax

BREAK

Remarks

The BREAK statement is a control statement that allows you to leave a loop. Execution resumes at the first statement after the loop.

Permissions

None.

Side effects

None.

See also

- "WHILE statement [T-SQL]" on page 718
- "CONTINUE statement [T-SQL]" on page 373
- "BEGIN statement" on page 351
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

In this example, the BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise, the loop continues until the average price is greater than or equal to $30:

```
WHILE ( SELECT AVG( UnitPrice ) FROM Products ) < $30
BEGIN
   UPDATE Products
   SET UnitPrice = UnitPrice + 2
   IF ( SELECT MAX(UnitPrice) FROM Products ) > $50
      BREAK
END
```


CALL statement

Use this statement to invoke a procedure.

Syntax 1

[*variable* =] **CALL** *procedure-name* **(** [*expression*, …] **)**

Syntax 2

[*variable* =] **CALL** *procedure-name* **(** [*parameter-name* = *expression*, …] **)**

Remarks

The CALL statement invokes a procedure that has been previously created with a CREATE PROCEDURE statement. When the procedure completes, any INOUT or OUT parameter value is copied back.

The argument list can be specified by position or by using keyword format. By position, the arguments will match up with the corresponding parameter in the parameter list for the procedure. By keyword, the arguments are matched up with the named parameters.

Procedure arguments can be assigned default values in the CREATE PROCEDURE statement, and missing parameters are assigned the default value. If no default is set, and an argument is not provided, an error is given.

Inside a procedure, a CALL statement can be used in a DECLARE statement when the procedure returns result sets. See “Returning results from procedures” [*SQL Anywhere Server - SQL Usage*].

Procedures can return an integer value (as a status indicator, say) using the RETURN statement. You can save this return value in a variable using the equality sign as an assignment operator:

```
CREATE VARIABLE returnval INT;
returnval = CALL proc_integer ( arg1 = val1, ... )
```

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

- “CREATE FUNCTION statement” on page 399
- “CREATE PROCEDURE statement” on page 414
- “GRANT statement” on page 548
- “EXECUTE statement [T-SQL]” on page 517
- “Using Procedures, Triggers, and Batches” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

Call the ShowCustomers procedure. This procedure has no parameters, and returns a result set.


```
CALL ShowCustomers();
```

The following Interactive SQL example creates a procedure to return the number of orders placed by the customer whose ID is supplied, creates a variable to hold the result, calls the procedure, and displays the result.

```
CREATE PROCEDURE OrderCount (IN customer_ID INT, OUT Orders INT)
BEGIN
   SELECT COUNT(SalesOrders.ID)
   INTO Orders
   FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
   WHERE Customers.ID = customer_ID;
END
go

 -- Create a variable to hold the result
CREATE VARIABLE Orders INT
go
-- Call the procedure, FOR customer 101
CALL OrderCount ( 101, Orders )
go
--  Display the result
SELECT Orders FROM DUMMY
go
```


CASE statement

Use this statement to select an execution path based on multiple cases.

Syntax 1

CASE *value-expression*
WHEN [*constant* | **NULL**] **THEN** *statement-list* …
[**WHEN** [*constant* | **NULL**] **THEN** *statement-list*] …
[**ELSE** *statement-list*]
END CASE

Syntax 2

CASE
WHEN [*search-condition* | **NULL**] **THEN** *statement-list* …
[**WHEN** [*search-condition* | **NULL**] **THEN** *statement-list*] …
[**ELSE** *statement-list*]
END CASE

Remarks

Syntax 1 The CASE statement is a control statement that allows you to choose a list of SQL statements to execute based on the value of an expression. The *value-expression* is an expression that takes on a single value, which may be a string, a number, a date, or other SQL data type. If a WHEN clause exists for the value of *value-expression*, the *statement-list* in the WHEN clause is executed. If no appropriate WHEN clause exists, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed. Execution resumes at the first statement after the END CASE.

If the *value-expression* can be null, use the ISNULL function to replace the NULL *value-expression* with a different expression.

Syntax 2 With this form, the statements are executed for the first satisfied *search-condition* in the CASE statement. The ELSE clause is executed if none of the *search-conditions* are met.

If the expression can be NULL, use the following syntax for the first *search-condition*:

```
WHEN search-condition IS NULL THEN statement-list
```

CASE statement is different from CASE expression
Do not confuse the syntax of the CASE statement with that of the CASE expression. See "CASE expressions" on page 17.

Permissions

None.

Side effects

None.

See also

- "ISNULL function [Miscellaneous]" on page 186
- "Unknown Values: NULL" [*SQL Anywhere Server - SQL Usage*]

- "BEGIN statement" on page 351
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following procedure using a case statement classifies the products listed in the Products table of the SQL Anywhere sample database into one of shirt, hat, shorts, or unknown.

```
CREATE PROCEDURE ProductType (IN product_ID INT, OUT type CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20);
   SELECT Name INTO prod_name FROM Products
   WHERE ID = product_ID;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
   WHEN 'Shorts' THEN
      SET type = 'Shorts'
   ELSE
      SET type = 'UNKNOWN'
   END CASE;
END
```

The following example uses Syntax 2 to generate a message about product quantity within the SQL Anywhere sample database.

```
CREATE PROCEDURE StockLevel (IN product_ID INT)
BEGIN
   DECLARE qty INT;
   SELECT Quantity INTO qty FROM Products
   WHERE ID = product_ID;
   CASE
   WHEN qty < 30 THEN
      MESSAGE 'Order Stock' TO CLIENT;
   WHEN qty > 100 THEN
      MESSAGE 'Overstocked' TO CLIENT;
   ELSE
      MESSAGE 'Sufficient stock on hand' TO CLIENT;
   END CASE;
END
```


CHECKPOINT statement

Use this statement to checkpoint the database.

Syntax

CHECKPOINT

Remarks

The CHECKPOINT statement forces the database server to execute a checkpoint. Checkpoints are also performed automatically by the database server according to an internal algorithm. It is not normally required for applications to issue the CHECKPOINT statement.

Permissions

DBA authority is required to checkpoint the network database server.

No permissions are required to checkpoint the personal database server.

Side effects

None.

See also

- "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*]
- "checkpoint_time option [database]" [*SQL Anywhere Server - Database Administration*]
- "recovery_time option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

CLEAR statement [Interactive SQL]

Use this statement to clear the Interactive SQL panes.

Syntax

CLEAR

Remarks

The CLEAR statement is used to clear the SQL Statements pane, the Messages pane and the Results, Messages, and Plan tabs in the Results pane.

Permissions

None.

Side effects

Closes the cursor associated with the data being cleared.

Standards and compatibility

- **SQL/2003** Vendor extension.

CLOSE statement [ESQL] [SP]

Use this statement to close a cursor.

Syntax

CLOSE *cursor-name*

cursor-name : *identifier* | *hostvar*

Remarks

This statement closes the named cursor.

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

- "OPEN statement [ESQL] [SP]" on page 601
- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "PREPARE statement [ESQL]" on page 610

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following examples close cursors in embedded SQL.

```
EXEC SQL CLOSE employee_cursor;
EXEC SQL CLOSE :cursor_var;
```

The following procedure uses a cursor.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
      FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
   SELECT CompanyName, CAST(    sum(SalesOrderItems.Quantity *
   Products.UnitPrice) AS INTEGER) VALUE
   FROM Customers
   LEFT OUTER JOIN SalesOrders
   LEFT OUTER JOIN SalesOrderItems
   LEFT OUTER JOIN Products
   GROUP BY CompanyName;

DECLARE ThisValue INT;
   DECLARE ThisCompany CHAR(35);
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
```



```
            FETCH NEXT curThisCust
            INTO ThisCompany, ThisValue;
               IF SQLSTATE = err_notfound THEN
                  LEAVE CustomerLoop;
               END IF;
               IF ThisValue > TopValue THEN
                  SET TopValue = ThisValue;
                  SET TopCompany = ThisCompany;
               END IF;
            END LOOP CustomerLoop;
         CLOSE curThisCust;
      END
```


COMMENT statement

Use this statement to store a comment for a database object in the system tables.

Syntax

COMMENT ON
{
  COLUMN [*owner*.]*table-name*.*column-name*
 | **EVENT** *event-name*
 | **FOREIGN KEY** [*owner*.]*table-name*.*role-name*
 | **INDEX** [[*owner*.] *table*.]*index-name*
 | **JAVA CLASS** *java-class-name*
 | **JAVA JAR** *java-jar-name*
 | **INTEGRATED LOGIN** *integrated-login-id*
 | **PROCEDURE** [*owner*.]*procedure-name*
 | **SERVICE** *web-service-name*
 | **TABLE** [*owner*.]*table-name*
 | **TRIGGER** [[*owner*.]*tablename*.]*trigger-name*
 | **USER** *userid*
 | **VIEW** [*owner*.]*view-name*
 | **MATERIALIZED VIEW** [*owner*.]*materialized-view-name*
 | **PRIMARY KEY ON** [*owner*.]*table-name*
 | **KERBEROS LOGIN "***client-Kerberos-principal***"**
}
IS *comment*

comment : *string* | **NULL**

Remarks

The COMMENT statement allows you to set a remark (comment) for an object in the database. The COMMENT statement updates remarks listed in the ISYSREMARKS system table. You can remove a comment by setting it to NULL. For a comment on an index or trigger, the owner of the comment is the owner of the table on which the index or trigger is defined.

You cannot add comments for local temporary tables.

Permissions

Must either be the owner of the database object being commented, or have DBA authority.

Side effects

Automatic commit.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following examples show how to add and remove a comment.

Add a comment to the Employees table.


```
COMMENT
ON TABLE Employees
IS 'Employee information';
```

Remove the comment from the Employees table.

```
COMMENT
ON TABLE Employees
IS NULL;
```

To view the comment set for an object, use a SELECT statement similar to the following. This example retrieves the comment set for the ViewSalesOrders view in the SQL Anywhere sample database.

```
SELECT remarks
FROM SYSTAB t, SYSREMARK r
WHERE t.object_id = r.object_id
AND t.table_name = 'ViewSalesOrders';
```


COMMIT statement

Use this statement to make changes to the database permanent, or to terminate a user-defined transaction.

Syntax 1

COMMIT [**WORK**]

Syntax 2

COMMIT TRAN[**SACTION**] [*transaction-name*]

Parameters

transaction-name An optional name assigned to this transaction. It must be a valid identifier. You should use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

For more information on transaction nesting in Adaptive Server Enterprise and SQL Anywhere, see "BEGIN TRANSACTION statement [T-SQL]" on page 354. For more information on savepoints, see "SAVEPOINT statement" on page 647.

You can use a set of options to control the detailed behavior of the COMMIT statement. See:

- "cooperative_commit_timeout option [database]" [*SQL Anywhere Server - Database Administration*]
- "cooperative_commits option [database]" [*SQL Anywhere Server - Database Administration*]
- "delayed_commits option [database]" [*SQL Anywhere Server - Database Administration*]
- "delayed_commit_timeout option [database]" [*SQL Anywhere Server - Database Administration*]

You can use the Commit connection property to return the number of Commits on the current connection. See "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

Remarks

Syntax 1 The COMMIT statement ends a transaction and makes all changes made during this transaction permanent in the database.

Data definition statements all carry out a commit automatically. For information, see the Side effects listing for each SQL statement.

The COMMIT statement fails if the database server detects any invalid foreign keys. This makes it impossible to end a transaction with any invalid foreign keys. Usually, foreign key integrity is checked on each data manipulation operation. However, if the database option wait_for_commit is set On or a particular foreign key was defined with a CHECK ON COMMIT option, the database server delays integrity checking until the COMMIT statement is executed.

Syntax 2 You can use BEGIN TRANSACTION and COMMIT TRANSACTION statements in pairs to construct nested transactions. Nested transactions are similar to savepoints. When executed as the outermost of a set of nested transactions, the statement makes changes to the database permanent. When executed inside a transaction, the COMMIT TRANSACTION statement decreases the nesting level of transactions by one. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Syntax 2 is a Transact-SQL extension.

Permissions

None.

Side effects

Closes all cursors except those opened WITH HOLD.

Deletes all rows of declared temporary tables on this connection, unless they were declared using ON COMMIT PRESERVE ROWS.

See also

- "SAVEPOINT statement" on page 647
- "BEGIN TRANSACTION statement [T-SQL]" on page 354
- "PREPARE TO COMMIT statement" on page 612
- "ROLLBACK statement" on page 642

Standards and compatibility

- **SQL/2003** Syntax 1 is a core feature. Syntax 2 is a Transact-SQL extension.

Example

The following statement commits the current transaction:

```
COMMIT;
```

The following Transact-SQL batch reports successive values of @@trancount as 0, 1, 2, 1, 0.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
go
```


CONFIGURE statement [Interactive SQL]

Use this statement to open the Interactive SQL Options dialog.

Syntax

CONFIGURE

Remarks

The CONFIGURE statement opens the Interactive SQL Options dialog. This window displays the current settings of all Interactive SQL options. It does not display or allow you to modify database options. You can configure Interactive SQL settings in this dialog.

Permissions

None.

Side effects

None.

See also

- "SET OPTION statement" on page 664

Standards and compatibility

- **SQL/2003** Vendor extension.

CONNECT statement [ESQL] [Interactive SQL]

Use this statement to establish a connection to a database.

Syntax 1

CONNECT
[**TO** *database-server-name*]
[**DATABASE** *database-name*]
[**AS** *connection-name*]
[**USER**] *userid* [**IDENTIFIED BY** *password*]

database-server-name, *database-name*, *connection-name*, *userid*, *password* :
{ *identifier* | *string* | *hostvar* }

Syntax 2

CONNECT USING *connect-string*

connect-string : { *identifier* | *string* | *hostvar* }

Parameters

AS clause A connection can optionally be named by specifying the AS clause. This allows multiple connections to the same database, or multiple connections to the same or different database servers, all simultaneously. Each connection has its own associated transaction. You may even get locking conflicts between your transactions if, for example, you try to modify the same record in the same database from two different connections.

Syntax 2 A *connect-string* is a list of parameter settings of the form keyword=*value*, separated by semicolons, and must be enclosed in single quotes.

For more information on connection strings, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Remarks

The CONNECT statement establishes a connection to the database identified by *database-name* running on the database server identified by *database-server-name*. This statement is not supported in procedures, triggers, events, or batches.

Embedded SQL behavior In embedded SQL, if no *database-server-name* is specified, the default local database server is assumed (the first database server started). If no *database-name* is specified, the first database on the given server is assumed.

The WHENEVER statement, SET SQLCA and some DECLARE statements do not generate code and thus may appear before the CONNECT statement in the source file. Otherwise, no statements are allowed until a successful CONNECT statement has been executed.

The user ID and password are used for permission checks on all dynamic SQL statements.

For a detailed description of the connection algorithm, see "Troubleshooting connections" [*SQL Anywhere Server - Database Administration*].

Note
For SQL Anywhere, only Syntax 1 is valid with embedded SQL. For UltraLite, both Syntax 1 and Syntax 2 can be used with embedded SQL.

Interactive SQL behavior If no database or server is specified in the CONNECT statement, Interactive SQL remains connected to the current database, rather than to the default server and database. If a database name is specified without a server name, Interactive SQL attempts to connect to the specified database on the current server. If a server name is specified without a database name, Interactive SQL connects to the default database on the specified server.

For example, if the following batch is executed while connected to a database, the two tables are created in the same database.

```
CREATE TABLE t1( c1 int );
CONNECT DBA IDENTIFIED BY sql;
CREATE TABLE t2 (c1 int );
```

No other database statements are allowed until a successful CONNECT statement has been executed.

When Interactive SQL is run in windowed mode, you are prompted for any missing connection parameters.

When Interactive SQL is running in command-prompt mode (-nogui is specified when you start Interactive SQL from a command prompt) or batch mode, or if you execute CONNECT without an AS clause, an unnamed connection is opened. If there is another unnamed connection already opened, the old one is automatically closed. Otherwise, existing connections are not closed when you run CONNECT.

Multiple connections are managed through the concept of a current connection. After a successful connect statement, the new connection becomes the current one. To switch to a different connection, use the SET CONNECTION statement. The DISCONNECT statement is used to drop connections.

When connecting to Interactive SQL, specifying CONNECT [USER] *userid* is the same as executing a SETUSER WITH OPTION *userid* statement. See "SETUSER statement" on page 671.

In Interactive SQL, the connection information (including the database name, your user ID, and the database server) appears in the title bar above the SQL Statements pane. If you are not connected to a database, Not Connected appears in the title bar.

Note
Both Syntax 1 and Syntax 2 are valid with Interactive SQL with the exception that Interactive SQL does not support the *hostvar* argument.

Permissions

None.

Side effects

None.

See also

- "GRANT statement" on page 548

- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 497
- "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 661
- "SETUSER statement" on page 671
- "Connection parameters" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Syntax 1 is a SQL/foundation feature outside of core SQL. Syntax 2 is a vendor extension.

Examples

The following are examples of CONNECT usage within embedded SQL.

```
EXEC SQL CONNECT AS :conn_name
USER :userid IDENTIFIED BY :password;
EXEC SQL CONNECT USER "DBA" IDENTIFIED BY "sql";
```

The following examples assume that the SQL Anywhere sample database has already been started.

Connect to a database from Interactive SQL. Interactive SQL prompts for a user ID and a password.

```
CONNECT
```

Connect to the default database as DBA from Interactive SQL. Interactive SQL prompts for a password.

```
CONNECT USER "DBA"
```

Connect to the sample database as user DBA from Interactive SQL.

```
CONNECT
TO demo10
USER DBA
IDENTIFIED BY sql
```

Connect to the sample database using a connect string, from Interactive SQL.

```
CONNECT
USING 'UID=DBA;PWD=sql;DBN=demo'
```

Once you connect to the sample database, the database name, your user ID, and the database server name appear on the title bar: **demo** (DBA) on **demo10**.

CONTINUE statement [T-SQL]

Use this statement to restart a loop.

Syntax

CONTINUE [*statement-label*]

Remarks

The CONTINUE statement is a control statement that allows you to restart a loop. Execution continues at the first statement in the loop. When CONTINUE appears within a set of statements using the Watcom-SQL, *statement-label* must be specified.

When CONTINUE appears within a set of statements using the Transact-SQL, *statement-label* must not be used.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 595
- "WHILE statement [T-SQL]" on page 718
- "FOR statement" on page 530
- "BEGIN statement" on page 351
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following fragment shows how the CONTINUE statement is used to restart a loop. This example displays the odd numbers between 1 and 10.

```
BEGIN
   DECLARE i INT;
   SET i = 0;
   lbl:
   WHILE i < 10 LOOP
      SET i = i + 1;
      IF mod( i, 2 ) = 0 THEN
         CONTINUE lbl
      END IF;
      MESSAGE 'The value ' || i || ' is odd.' TO CLIENT;
   END LOOP lbl;
END
```


CREATE DATABASE statement

Use this statement to create a database. The database is stored as an operating system file.

Syntax

CREATE DATABASE *db-file-name-string*
  [**ACCENT** { **RESPECT** | **IGNORE** | **FRENCH** }]
  [**ASE** [**COMPATIBLE**]]
  [**BLANK PADDING** { **ON** | **OFF** }]
  [**CASE** { **RESPECT** | **IGNORE** }]
  [**CHECKSUM** { **ON** | **OFF** }]
  [**COLLATION** *collation-label*[**(** *collation-tailoring-string* **)**]]
  [**DATABASE SIZE** *size* { **KB** | **MB** | **GB** | **PAGES** | **BYTES** }]
  [**DBA USER** *userid*]
  [**DBA PASSWORD** *password*]
  [**ENCODING** *encoding-label*]
  [**ENCRYPTED** [**TABLE**] { *algorithm-key-spec* | **OFF** }]
  [**JCONNECT** { **ON** | **OFF** }]
  [**PAGE SIZE** *page-size*]
  [**NCHAR COLLATION** *nchar-collation-label*[**(** *collation-tailoring-string* **)**]]
  [[**TRANSACTION**] { **LOG OFF** | **LOG ON** [*log-file-name-string*]
    [**MIRROR** *mirror-file-name-string*]] }

page-size :
2048 | **4096** | **8192** | **16384** | **32768**

algorithm-key-spec:
ON
| [**ON**] **KEY** *key* [**ALGORITHM** { **'AES'** | **'AES_FIPS'** }]
| [**ON**] **ALGORITHM** { **'AES'** | **'AES_FIPS'** } **KEY** *key*
| [**ON**] **ALGORITHM 'SIMPLE'**

Parameters

The file names (*db-file-name-string*, *log-file-name-string*, and *mirror-file-name-string*) are strings containing operating system file names. As literal strings, they must be enclosed in single quotes.

- If you specify a path, any backslash characters (\) must be doubled if they are followed by an n or an x. Escaping them prevents them from being interpreted as new line characters (\n) or as hexadecimal numbers (\x), according to the rules for strings in SQL.

  It is always safer to escape the backslash character. For example,

  ```
  CREATE DATABASE 'c:\\databases\\my_db.db'
  LOG ON 'e:\\logdrive\\my_db.log';
  ```

- If you specify no path, or a relative path, the database file is created relative to the working directory of the database server. If you specify no path for a log file, the file is created in the same directory as the database file.

- If you provide no file extension, a file is created with extension *.db* for databases, *.log* for the transaction log, and *.mlg* for the mirror log.

ACCENT clause This clause is used to specify accent sensitivity for the database. Support for this clause is deprecated. Use the collation tailoring options provided for the COLLATION and NCHAR COLLATION clauses to specify accent sensitivity.

The ACCENT clause applies only when using the UCA (Unicode Collation Algorithm) for the collation specified in the COLLATION or NCHAR COLLATION clause. ACCENT RESPECT causes the UCA string comparison to respect accent differences between letters. For example, e is less than é. ACCENT FRENCH is similar to ACCENT RESPECT, except that accents are compared from right to left, consistent with the rules of the French language. ACCENT IGNORE causes string comparisons to ignore accents. For example, e is equal to é. This is the default behavior. For more information, see "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*].

ASE COMPATIBLE clause Do not create the SYS.SYSCOLUMNS and SYS.SYSINDEXES views. By default, these views are created for compatibility with system tables available in Watcom SQL (version 4 and earlier of this software). These views conflict with the Sybase Adaptive Server Enterprise compatibility views dbo.syscolumns and dbo.sysindexes.

BLANK PADDING clause
SQL Anywhere compares all strings as if they are varying length and stored using the VARCHAR domain. This includes string comparisons involving fixed length CHAR or NCHAR columns. In addition, SQL Anywhere never trims or pads values with trailing blanks when the values are stored in the database.

By default, SQL Anywhere treats blanks as significant characters. Hence the value 'a ' (the character 'a' followed by a blank) is not equivalent to the single-character string 'a'. Inequality comparisons also treat a blank as any other character in the collation.

If blank padding is enabled (specifying BLANK PADDING ON), the semantics of string comparisons more closely follow the ANSI/ISO SQL standard. With blank-padding enabled, SQL Anywhere ignores trailing blanks in any comparison.

In the example above, an equality comparison of 'a ' to 'a' in a blank-padded database returns TRUE. With a blank-padded database, fixed-length string values are padded with blanks when they are fetched by an application. Whether or not the application receives a string truncation warning on such an assignment is controlled by the ansi_blanks connection option. See "ansi_blanks option [compatibility]" [*SQL Anywhere Server - Database Administration*].

CASE clause This clause is used to specify case sensitivity for the database. Support for this clause is deprecated. Use the collation tailoring options provided for the COLLATION and NCHAR COLLATION clauses to specify case sensitivity.

CASE RESPECT causes case-sensitive string comparisons for all CHAR and NCHAR data types. Comparisons using UCA consider the case of a letter only if the base letters and accents are all equal. For all other collations, uppercase and lowercase letters are distinct. For example, a is less than A, which is less than b, and so on. CASE IGNORE causes case-insensitive string comparisons. Uppercase and lowercase letters are considered to be exactly equal. By default, comparisons are case insensitive. CASE RESPECT is provided for compatibility with the ISO/ANSI SQL standard. Identifiers in the database are always case insensitive, even in case-sensitive databases.

CHECKSUM clause Checksums are used to determine whether a database page has been modified on disk. When you create a database with checksums enabled, a checksum is calculated for each page just before it is written to disk. The next time the page is read from disk, the page's checksum is recalculated and

compared to the checksum stored on the page. If the checksums are different, then the page has been modified on disk and an error occurs. Databases created with checksums enabled can also be validated using checksums. You can check whether a database was created with checksums enabled by executing the following statement:

```
SELECT DB_PROPERTY ( 'Checksum' );
```

This query returns ON if checksums are turned on, otherwise, it returns OFF. Checksums are turned off by default, so if the CHECKSUM clause is omitted, OFF is applied.

Regardless of the setting of this clause, the database server always calculates checksums for critical pages.

See "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*], "sa_validate system procedure" on page 934, or "VALIDATE statement" on page 713.

COLLATION clause The collation specified by the COLLATION clause is used for sorting and comparison of character data types (CHAR, VARCHAR, and LONG VARCHAR). The collation provides character comparison and ordering information for the encoding (character set) being used. If the COLLATION clause is not specified, SQL Anywhere chooses a collation based on the operating system language and encoding.

The collation can be chosen from the list of collations that use the SQL Anywhere Collation Algorithm, or it can be the Unicode Collation Algorithm (UCA). If UCA is specified, you should also specify the ENCODING clause.

It is important to choose your collation carefully. It cannot be changed after the database has been created. See "Choosing collations" [*SQL Anywhere Server - Database Administration*].

Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of keyword=value pairs, assembled in parentheses, following the collation name. For example, `... CHAR COLLATION 'UCA (locale=es;case=respect;accent=respect)'`. If you specify the ACCENT or CASE clause as well as a collation tailoring string that contains settings for case and accent, the values of the ACCENT and CASE clauses are used as defaults only. Following is a table of the supported keywords, including their allowed alternate forms, and their allowed values.

If UCA is specified by itself, the default tailoring applied is equivalent to `'UCA (case=UpperFirst;accent=Respect;punct=Primary)'`.

Note
All of the collation tailoring options below are supported when specifying the UCA collation. For all other collations, only case sensitivity tailoring is supported.
Also, databases created with collation tailoring options cannot be started using a pre-10.0.1 database server.

Collation tailoring options

Keyword	Collation	Alternate forms	Allowed values
Locale	UCA	(none)	Any valid locale code. For example, en.

Keyword	Collation	Alternate forms	Allowed values
CaseSensitivity	All supported collations	CaseSensitive, Case	♦ **respect** Respect case differences between letters. For the UCA collation, this is equivalent to UpperFirst. For other collations, it depends on the collation itself. ♦ **ignore** Ignore case differences between letters. ♦ **UpperFirst** Always sort upper case first (Aa). ♦ **LowerFirst** Always sort lowercase first (aA).
AccentSensitivity	UCA	AccentSensitive, Accent	♦ **respect** Respect accent differences between letters. ♦ **ignore** Ignore accent differences between letters. ♦ **French** Respect accent sensitivity with French rules.
PunctuationSensitivity	UCA	PunctuationSensitive, Punct	♦ **ignore** Ignore differences in punctuation. ♦ **primary** Use first level sorting (consider letter, only). For example, a > b. ♦ **quaternary** Use fourth level sorting: consider letter first, then case, then accent, and then punctuation. For example, multiByte, multibyte, multi-byte, and multi-Byte, are sorted as: ♦ multiByte ♦ multibyte ♦ multi-Byte ♦ multi-byte You cannot specify quaternary with a case or accent insensitive database.

Keyword	Collation	Alternate forms	Allowed values
SortType	UCA	(none)	The type of sort to use. Possible values: ♦ phonebook ♦ traditional ♦ standard ♦ pinyin ♦ stroke ♦ direct ♦ posix ♦ big5han ♦ gb2312han For more information about these sort types, see Unicode Technical Standard #35, at http://www.unicode.org/reports/tr35/.

DATABASE SIZE clause Pre-allocating space for the database helps reduce the risk of running out of space on the drive the database is located on. As well, it can help improve performance by increasing the amount of data that can be stored in the database before the database server needs to grow the database, which can be a time-consuming operation. You can use KB, MB, GB, or PAGES to specify units of kilobytes, megabytes, gigabytes, or pages respectively.

DBA USER clause Use this clause to specify a DBA user for the database. When you use this clause, you can no longer connect to the database as the default DBA user. If you do not specify this clause, the default DBA user ID is created.

DBA PASSWORD clause You can specify a different password for the DBA database user. If you do not specify this clause, the default password (**sql**) is used for the DBA user.

ENCODING clause Most collations specified in the COLLATION clause dictate both the encoding (character set) and ordering. For those collations, the ENCODING clause should not be specified. However, if the value specified in the COLLATION clause is UCA (Unicode Collation Algorithm), use the ENCODING clause to specify a locale-specific encoding and get the benefits of the UCA for comparison and ordering. The ENCODING clause may specify UTF-8 or any single-byte encoding for CHAR data types. ENCODING may not specify a multibyte encoding other than UTF-8.

If COLLATION is set to UCA and ENCODING is not specified, then SQL Anywhere uses UTF-8. For more information on the recommended encodings and collations, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

For more information on how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

ENCRYPTED or ENCRYPTED TABLE clause Encryption makes stored data unreadable. Use the ENCRYPTED keyword (without TABLE) when you want to encrypt the entire database. Use the ENCRYPTED TABLE clause when you only want to enable table encryption. Enabling table encryption means that the tables that are subsequently created or altered using the ENCRYPTED clause are encrypted

using the settings you specified at database creation. See "Encrypting tables" [*SQL Anywhere Server - Database Administration*].

There are two levels of database and table encryption: simple and strong. Simple encryption is equivalent to obfuscation. The data is unreadable, but someone with cryptographic expertise could decipher the data. For simple encryption, specify ENCRYPTED ON ALGORITHM SIMPLE, or ENCRYPTED ALGORITHM SIMPLE, or specify the ENCRYPTED ON clause without specifying an algorithm or key.

With strong encryption, the data is unreadable, and virtually undecipherable. It is recommended that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase, and includes numbers, letters, and special characters. For strong encryption, you use the ALGORITHM clause to specify a 128-bit AES algorithm (either AES or AES_FIPS) and the KEY clause to specify an encryption key.

On Windows CE, the AES_FIPS algorithm is supported with ARM processors.

Caution
Protect your key! Be sure to store a copy of your key in a safe location. A lost key will result in a completely inaccessible database, from which there is no recovery.

For more information about strong database encryption, see "Strong encryption" [*SQL Anywhere Server - Database Administration*].

JCONNECT clause To allow the Sybase jConnect JDBC driver access to system catalog information, specify JCONNECT ON. This will install the system objects that provide jConnect support. Specify JCONNECT OFF if you want to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information. JCONNECT is ON by default.

NCHAR COLLATION clause The collation specified by the NCHAR COLLATION clause is used for sorting and comparison of national character data types (NCHAR, NVARCHAR, and LONG NVARCHAR). The collation provides character ordering information for the UTF-8 encoding (character set) used for national characters. If the NCHAR COLLATION clause is not specified, SQL Anywhere uses the Unicode Collation Algorithm (UCA). The only other allowed collation is UTF8BIN, which provides a binary ordering of all characters whose encoding is greater than 0x7E. See "Choosing collations" [*SQL Anywhere Server - Database Administration*].

Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of keyword=value pairs, assembled in a quoted string, following the collation name. For example, `... NCHAR COLLATION 'UCA (locale=es;case=respect;accent=respect)'`. If you specify the ACCENT or CASE clause as well as a collation tailoring string that contains settings for case and accent, the values of the ACCENT and CASE clauses are used as defaults only. The syntax for specifying these options is identical to the syntax defined for the COLLATION clause, above. See "Collation tailoring options" on page 376.

Note
All of the collation tailoring options are supported when specifying the UCA collation. For all other collations, only the case sensitivity tailoring option is supported.

Note
Databases created with collation tailoring options cannot be started using a pre-10.0.1 database server.

PAGE SIZE clause The page size for a database can be 2048, 4096, 8192, 16384, or 32768 bytes. The default page size is 4096 bytes. Large databases generally obtain performance benefits from a larger page size, but there can be additional overhead associated with large page sizes.

For example,

```
CREATE DATABASE 'c:\\databases\\my_db.db'
PAGE SIZE 4096;
```

Page size limit
The page size cannot be larger than the page size used by the current server. The server page size is taken from the first set of databases started or is set on the server command line using the -gp option.

TRANSACTION LOG clause The transaction log is a file where the database server logs all changes made to the database. The transaction log plays a key role in backup and recovery (see "The transaction log" [*SQL Anywhere Server - Database Administration*]), and in data replication.

The MIRROR clause of the TRANSACTION clause allows you to provide a file name if you want to use a transaction log mirror. A transaction log mirror is an identical copy of a transaction log, usually maintained on a separate device, for greater protection of your data. By default, SQL Anywhere does not use a mirrored transaction log.

Remarks

Creates a database file with the supplied name and attributes. This statement is not supported in procedures, triggers, events, or batches.

Permissions

The permissions required to execute this statement are set on the server command line, using the -gu option. The default setting is to require DBA authority.

The account under which the database server is running must have write permissions on the directories where files are created.

Side effects

An operating system file is created.

See also

- "ALTER DATABASE statement" on page 301
- "DROP statement" on page 498
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- "DatabaseKey connection parameter [DBKEY]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement creates a database file named *mydb.db* in the *C:* directory.

```
CREATE DATABASE 'C:\\mydb.db'
TRANSACTION LOG ON
CASE IGNORE
PAGE SIZE 2048
ENCRYPTED OFF
BLANK PADDING OFF;
```

The following statement creates a database using code page 1252 and uses the UCA for both CHAR and NCHAR data types. Accents and case are respected during comparison and sorting.

```
CREATE DATABASE 'c:\\uca.db'
COLLATION 'UCA'
ENCODING 'CP1252'
NCHAR COLLATION 'UCA'
ACCENT RESPECT
CASE RESPECT;
```

The following statement creates a database, *myencrypteddb.db*, that is encrypted using simple encryption:

```
CREATE DATABASE 'myencrypteddb.db'
ENCRYPTED ON;
```

The following statement creates a database, *mystrongencryptdb.db*, that is encrypted using the key gh67AB2 (strong encryption):

```
CREATE DATABASE 'mystrongencryptdb.db'
ENCRYPTED ON KEY 'gh67AB2';
```

The following statement creates a database, *mytableencryptdb.db*, with table encryption enabled using simple encryption. Notice the keyword TABLE inserted after ENCRYPTED to indicate table encryption instead of database encryption:

```
CREATE DATABASE 'mytableencryptdb.db'
ENCRYPTED TABLE ON;
```

The following statement creates a database, *mystrongencrypttabledb.db*, with table encryption enabled using the key gh67AB2 (strong encryption), and the AES_FIPS encryption algorithm:

```
CREATE DATABASE 'mystrongencrypttabledb.db'
ENCRYPTED TABLE ON KEY 'gh67AB2'
ALGORITHM 'AES_FIPS';
```

The following statement creates a database file named *mydb.db* that uses collation 1252LATIN1. The NCHAR collation is set to UCA, with the locale set to es, and has case sensitivity and accent sensitivity enabled:

```
CREATE DATABASE 'my2.db'
      COLLATION '1252LATIN1(case=respect)'
      NCHAR COLLATION 'UCA(locale=es;case=respect;accent=respect)'
```


CREATE DBSPACE statement

Use this statement to define a new database space and create the associated database file.

Syntax

CREATE DBSPACE *dbspace-name* **AS** *file-name*

Parameters

dbspace-name An internal name for the database file. The *file-name* parameter is the actual name of the database file, with a path where necessary. You cannot use the following names for a dbspace because they are used for the pre-defined dbspaces: SYSTEM, TEMPORARY, TEMP, TRANSLOG, and TRANSLOGMIRROR. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

file-name A *file-name* without an explicit directory is created in the same directory as the main database file. Any relative directory is relative to the main database file. The *file-name* is relative to the database server. When you are using the database server for NetWare, the *file-name* should use a volume name (not a drive letter) when an absolute directory is specified.

Remarks

The CREATE DBSPACE statement creates a new database file. When a database is created, it is composed of one file. All tables and indexes created are placed in that file. CREATE DBSPACE adds a new file to the database. This file can be on a different disk drive than the main file, which means that the database can be larger than one physical device.

For each database, there is a limit of twelve dbspaces in addition to the main file.

Each table is contained entirely within one database file. The IN clause of the CREATE TABLE statement specifies the dbspace into which a table is placed. Tables are put into the main database file by default. You can also specify which dbspace tables are created in by setting the default_dbspace option before you create the tables.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Automatic checkpoint.

See also

- "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*]
- "DROP statement" on page 498
- "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Create a dbspace called library to hold the LibraryBooks table and its indexes.


```
CREATE DBSPACE library
AS 'c:\\library.db';
CREATE TABLE LibraryBooks (
  title char(100),
  author char(50),
  isbn char(30),
) IN library;
```


CREATE DECRYPTED FILE statement

This statement decrypts strongly encrypted databases.

Syntax

CREATE DECRYPTED FILE *newfile*
FROM *oldfile* **KEY** *key*

Parameters

FROM Lists the file name of the encrypted file.

KEY Lists the key required to access the encrypted file.

Remarks

This statement decrypts an encrypted database, transaction log file, or dbspace and creates a new, unencrypted file. The original file must be strongly encrypted using an encryption key. The resulting file is an exact copy of the encrypted file, without encryption and therefore requiring no encryption key.

If a database is decrypted using this statement, the corresponding transaction log file (and any dbspaces) must also be decrypted to use the database.

If a database requiring recovery is decrypted, its transaction log file must also be decrypted and recovery on the new database will still be necessary.

The name of the transaction log file remains the same in this process, so if the database and transaction log file are renamed, then you need to run dblog -t on the resulting database.

If you want to encrypt an existing database, you need to either use the CREATE ENCRYPTED FILE statement, or unload and reload the database using the dbunload -an option with either -ek or -ep. You can also use this method to change an existing encryption key.

You cannot use this statement on a database on which table encryption is enabled. If you have tables you want to decrypt, use the NOT ENCRYPTED clause of the ALTER TABLE statements to decrypt them. See "ALTER TABLE statement" on page 332.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

Must be a user with DBA authority.

Side effects

None.

See also

- "CREATE ENCRYPTED FILE statement" on page 388

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example decrypts the contacts database and creates a new unencrypted database called contacts2.

```
CREATE DECRYPTED FILE 'contacts2.db'
FROM 'contacts.db'
KEY 'Sd8f6654*Mnn';
```


CREATE DOMAIN statement

Use this statement to create a domain in a database.

Syntax

CREATE { **DOMAIN** | **DATATYPE** } [**AS**] *domain-name data-type*
[[**NOT**] **NULL**]
[**DEFAULT** *default-value*]
[**CHECK (** *condition* **)**]

domain-name : *identifier*

data-type : *built-in data type*, *with precision and scale*

Parameters

DOMAIN | DATATYPE It is recommended that you use CREATE DOMAIN, rather than CREATE DATATYPE, because CREATE DOMAIN is the ANSI/ISO SQL3 term.

NULL
This clause allows you to specify the nullability of a domain. When a domain is used to define a column, nullability is determined as follows:

- Nullability specified in the column definition
- Nullability specified in the domain definition
- If the nullability was not explicitly specified in either the column definition or the domain definition, then the setting of the allow_nulls_by_default option is used.

CHECK clause When creating a CHECK condition, you can use a variable name prefixed with the @ sign in the condition. When the data type is used in the definition of a column, such a variable is replaced by the column name. This allows CHECK conditions to be defined on data types and used by columns of any name.

Remarks

Domains are aliases for built-in data types, including precision and scale values where applicable. They improve convenience and encourage consistency in the database.

Domains are objects within the database. Their names must conform to the rules for identifiers. Domain names are always case insensitive, as are built-in data type names.

The user who creates a data type is automatically made the owner of that data type. No owner can be specified in the CREATE DATATYPE statement. The domain name must be unique, and all users can access the data type without using the owner as prefix.

Domains can have CHECK conditions and DEFAULT values, and you can indicate whether the data type permits NULL values or not. These conditions and values are inherited by any column defined on the data type. Any conditions or values explicitly specified on the column override those specified for the data type.

To drop the data type from the database, use the DROP statement. You must be either the owner of the data type or have DBA authority to drop a domain.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "DROP statement" on page 498
- "SQL Data Types" on page 47

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Example

The following statement creates a data type named address, which holds a 35-character string, and which may be NULL.

```
CREATE DOMAIN address CHAR( 35 ) NULL;
```

The following statement creates a data type named ID, which does not allow NULLS, and which is autoincremented by default.

```
CREATE DOMAIN ID INT
NOT NULL
DEFAULT AUTOINCREMENT;
```


CREATE ENCRYPTED FILE statement

This statement encrypts unencrypted databases, transaction logs, or dbspaces. It can also be used to change the encryption key for an encrypted database, or for a database with table encryption enabled.

Syntax

CREATE ENCRYPTED FILE *newfile*
FROM *oldfile*
{ **KEY** *key* | **KEY** *key* **OLD KEY** *oldkey* }
[**ALGORITHM** { **'AES'** | **'AES_FIPS'** }]

Parameters

FROM clause Specifies the name of the existing file (*oldfile*) on which to execute the CREATE ENCRYPTED FILE statement.

KEY clause Specifies the encryption key to use.

OLD KEY clause Specifies the current key with which the file is encrypted.

ALGORITHM clause Specifies the algorithm used to encrypt the file. If you do not specify an algorithm, AES is used by default.

Remarks

Use the CREATE ENCRYPTED FILE statement to:

- take an unencrypted database, transaction log, or dbspace and create a new file encrypted with the specified key
- take an encrypted database, transaction log, or dbspace and create a new file encrypted with a new encryption key

The CREATE ENCRYPTED FILE statement produces a new file (*newfile*), and does not replace or remove the previous version of the file (*oldfile*).

If a database is encrypted using this statement, you must encrypt the corresponding transaction log file (and any dbspaces) using the same encryption key to use the database. You cannot mix encrypted and unencrypted files, nor can you mix encrypted files with different encryption algorithms or different keys.

If a database requiring recovery is encrypted, its transaction log file must also be encrypted and recovery on the new database will still be necessary.

The name of the transaction log file remains the same in this process, so if the database and transaction log file are renamed, then you need to run dblog -t on the resulting database.

You can also encrypt an existing database or change an existing encryption key by unloading and reloading the database using the dbunload -an option with either -ek or -ep.

If you have a database on which table encryption is enabled, you cannot encrypt the database using this statement. However, you can use this statement to change the key used for table encryption.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

Must be a user with DBA authority.

On Windows CE, the FIPS algorithm is only supported with ARM processors.

Side effects

None.

See also

- "Encrypting a database" [*SQL Anywhere Server - Database Administration*]
- "CREATE DECRYPTED FILE statement" on page 384
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example encrypts the contacts database and creates a new database called contacts2 that is encrypted with AES_FIPS encryption.

```
CREATE ENCRYPTED FILE 'contacts2.db'
FROM 'contacts.db'
KEY 'Sd8f6654*Mnn'
ALGORITHM AES_FIPS;
```

The following example encrypts the contacts database and the contacts log file, renaming the both files. You will need to `run dblog -ek abcd -t contacts2.log contacts.db`, since the log has been renamed and the database file still points to the old log.

```
CREATE ENCRYPTED FILE 'contacts2.db'
FROM 'contacts.db'
KEY 'Sd8f6654*Mnn'
CREATE ENCRYPTED FILE 'contacts2.log'
FROM 'contacts.db'
KEY 'Te9g7765*Noo';
```

The following example encrypts the contacts database and the contacts log file, leaving the original log file name untouched. In this case, you do not need to run dblog, since the name of the file remains the same.

```
CREATE ENCRYPTED FILE 'newpath\contacts.db'
FROM 'contacts.db'
KEY 'Sd8f6654*Mnn'
CREATE ENCRYPTED FILE 'newpath\contacts.log'
FROM 'contacts.log'
KEY 'Sd8f6654*Mnn';
```

The following example changes the encryption key of the contacts database.

```
CREATE ENCRYPTED FILE 'newcontacts.db'
 FROM 'contacts.db'
 KEY 'newkey' OLD KEY 'oldkey';
 DEL contacts.db
 RENAME newcontacts.db contacts.db;
```


CREATE EVENT statement

Use this statement to define an event and its associated handler for automating predefined actions. Also, to define scheduled actions.

Syntax

CREATE EVENT *event-name*
[**TYPE** *event-type*
  [**WHERE** *trigger-condition* [**AND** *trigger-condition*] …]
  | **SCHEDULE** *schedule-spec*, …]
[**ENABLE** | **DISABLE**]
[**AT** { **CONSOLIDATED** | **REMOTE** | **ALL** }]
[**HANDLER**
  BEGIN
…
  END]

event-type :
BackupEnd | **"Connect"**
| **ConnectFailed** | **DatabaseStart**
| **DBDiskSpace** | **"Disconnect"**
| **GlobalAutoincrement** | **GrowDB**
| **GrowLog** | **GrowTemp**
| **LogDiskSpace** | **MirrorFailover**
| **MirrorServerDisconnect** | **"RAISERROR"**
| **ServerIdle** | **TempDiskSpace**

trigger-condition :
event_condition(*condition-name* **)** { **=** | **<** | **>** | **!=** | **<=** | **>=** } *value*

schedule-spec :
[*schedule-name*]
  { **START TIME** *start-time* | **BETWEEN** *start-time* **AND** *end-time* }
  [**EVERY** *period* { **HOURS** | **MINUTES** | **SECONDS** }]
  [**ON** { **(** *day-of-week*, … **)** | **(** *day-of-month*, … **)** }]
  [**START DATE** *start-date*]

event-name | *schedule-name* : *identifier*

day-of-week : *string*

day-of-month | *value* | *period* : *integer*

start-time | *end-time* : *time*

start-date : *date*

Parameters

CREATE EVENT clause The event name is an identifier. An event has a creator, which is the user creating the event, and the event handler executes with the permissions of that creator. This is the same as stored procedure execution. You cannot create events owned by other users.

TYPE clause You can specify the TYPE clause with an optional WHERE clause; or specify the SCHEDULE.

The *event-type* is one of the listed set of system-defined event types. The event types are case insensitive. To specify the conditions under which this *event-type* triggers the event, use the WHERE clause. For a description of event-types not listed below, see "Understanding system events" [*SQL Anywhere Server - Database Administration*].

- **DiskSpace event types** If the database contains an event handler for one of the DiskSpace types, the database server checks the available space on each device associated with the relevant file every 30 seconds.

  In the event the database has more than one dbspace, on separate drives, DBDiskSpace checks each drive and acts depending on the lowest available space.

  The LogDiskSpace event type checks the location of the transaction log and any mirrored transaction log, and reports based on the least available space.

  Disk space event types are not supported on Windows CE.

  The TempDiskSpace event type checks the amount of temporary disk space.

  If the appropriate event handlers have been defined (DBDiskSpace, LogDiskSpace, or TempDiskSpace), the database server checks the available space on each device associated with a database file every 30 seconds. Similarly, if an event has been defined to handle the system event type ServerIdle, the database server notifies the handler when no requests have been process during the previous 30 seconds.

  You can specify the -fc option when starting the database server to implement a callback function when the database server encounters a file system full condition.

  See "-fc server option" [*SQL Anywhere Server - Database Administration*].

- **GlobalAutoIncrement event type** The event fires on *each* insert when the number of remaining values for a GLOBAL AUTOINCREMENT is less than 1% of the end of its range. A typical action for the handler could be to request a new value for the global_database_id option, based on the table and number of remaining values which are supplied as parameters to this event.

  You can use the event_condition function with RemainingValues as an argument for this event type.

- **ServerIdle event type** If the database contains an event handler for the **ServerIdle** type, the database server checks for server activity every 30 seconds.

- **Database mirroring event types** The MirrorServerDisconnect event fires when a connection from the primary database server to the mirror server or arbiter server is lost, and the MirrorFailover event fires whenever a server takes ownership of the database. See "Database mirroring system events" [*SQL Anywhere Server - Database Administration*].

WHERE clause The trigger condition determines the condition under which an event is fired. For example, to take an action when the disk containing the transaction log becomes more than 80% full, use the following triggering condition:


```
...
WHERE event_condition( 'LogDiskSpacePercentFree' ) < 20
...
```

The argument to the event_condition function must be valid for the event type.

You can use multiple AND conditions to make up the WHERE clause, but you cannot use OR conditions or other conditions.

For information on valid arguments, see "EVENT_CONDITION function [System]" on page 158.

SCHEDULE clause This clause specifies when scheduled actions are to take place. The sequence of times acts as a set of triggering conditions for the associated actions defined in the event handler.

You can create more than one schedule for a given event and its associated handler. This permits complex schedules to be implemented. While it is compulsory to provide a schedule-name when there is more than one schedule, it is optional if you provide only a single schedule.

A scheduled event is recurring if its definition includes EVERY or ON; if neither of these reserved words is used, the event will execute at most once. An attempt to create a non-recurring scheduled event for which the start time has passed will generate an error. When a non-recurring scheduled event has passed, its schedule is deleted, but the event handler is not deleted.

Scheduled event times are calculated when the schedules are created, and again when the event handler completes execution. The next event time is computed by inspecting the schedule or schedules for the event, and finding the next schedule time that is in the future. If an event handler is instructed to run every hour between 9:00 and 5:00, and it takes 65 minutes to execute, it runs at 9:00, 11:00, 1:00, 3:00, and 5:00. If you want execution to overlap, you must create more than one event.

The subclauses of a schedule definition are as follows:

- **START TIME** The first scheduled time for each day on which the event is scheduled. If a START DATE is specified, the START TIME refers to that date. If no START DATE is specified, the START TIME is on the current day (unless the time has passed) and each subsequent day (if the schedule includes EVERY or ON).
- **BETWEEN … AND** A range of times during the day outside of which no scheduled times occur. If a START DATE is specified, the scheduled times do not occur until that date.
- **EVERY** An interval between successive scheduled events. Scheduled events occur only after the START TIME for the day, or in the range specified by BETWEEN … AND.
- **ON** A list of days on which the scheduled events occur. The default is every day if EVERY is specified. Days can be specified as days of the week or days of the month.

  Days of the week are Mon, Tues, and so on. You may also use the full forms of the day, such as Monday. You must use the full forms of the day names if the language you are using is not English, is not the language requested by the client in the connection string, and is not the language which appears in the Server Messages window.

  Days of the month are integers from 0 to 31. A value of 0 represents the last day of any month.

- **START DATE** The date on which scheduled events are to start occurring. The default is the current date.

Each time a scheduled event handler is completed, the next scheduled time and date is calculated.

1. If the EVERY clause is used, find whether the next scheduled time falls on the current day, and is before the end of the BETWEEN … AND range. If so, that is the next scheduled time.
2. If the next scheduled time does not fall on the current day, find the next date on which the event is to be executed.
3. Find the START TIME for that date, or the beginning of the BETWEEN … AND range.

ENABLE | DISABLE By default, event handlers are enabled. When DISABLE is specified, the event handler does not execute even when the scheduled time or triggering condition occurs. A TRIGGER EVENT statement does *not* cause a disabled event handler to be executed.

AT clause If you want to execute events at remote or consolidated databases in a SQL Remote setup, you can use this clause to restrict the databases at which the event is handled. By default, all databases execute the event.

HANDLER clause Each event has one handler.

Remarks

Events can be used in two main ways:

- **Scheduling actions** The database server carries out a set of actions on a schedule of times. You could use this capability to schedule backups, validity checks, queries to fill up reporting tables, and so on.
- **Event handling actions** The database server carries out a set of actions when a predefined event occurs. The events that can be handled include disk space restrictions (when a disk fills beyond a specified percentage), when the database server is idle, and so on. The actions of an event handler are committed if no error is detected during execution, and rolled back if errors are detected.

An event definition includes two distinct pieces. The trigger condition can be an occurrence, such as a disk filling up beyond a defined threshold. A schedule is a set of times, each of which acts as a trigger condition. When a trigger condition is satisfied, the event handler executes. The event handler includes one or more actions specified inside a compound statement (BEGIN… END).

If no trigger condition or schedule specification is supplied, only an explicit TRIGGER EVENT statement can trigger the event. During development, you may want to test event handlers using TRIGGER EVENT, and add the schedule or WHERE clause once testing is complete.

Event errors are logged to the database server console.

After each execution of an event handler, a COMMIT occurs if no errors occurred. A ROLLBACK occurs if there was an error.

When event handlers are triggered, the database server makes context information, such as the connection ID that caused the event to be triggered, available to the event handler using the event_parameter function. For more information about event_parameter, see "EVENT_PARAMETER function [System]" on page 160.

Permissions

Must have DBA authority.

Event handlers execute on a separate connection, with the permissions of the event owner. To execute with authority other than DBA, you can call a procedure from within the event handler: the procedure executes with the permissions of its owner. The separate connection does not count towards the ten-connection limit of the personal database server.

Side effects

Automatic commit.

See also

- "BEGIN statement" on page 351
- "ALTER EVENT statement" on page 308
- "COMMENT statement" on page 365
- "DROP statement" on page 498
- "TRIGGER EVENT statement" on page 692
- "EVENT_PARAMETER function [System]" on page 160
- "Understanding system events" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Instruct the database server to carry out an automatic backup to tape using the first tape drive, every day at 1 A.M.

```
CREATE EVENT DailyBackup
SCHEDULE daily_backup
START TIME '1:00AM' EVERY 24 HOURS
HANDLER
   BEGIN
      BACKUP DATABASE TO '\\\\.\\tape0'
      ATTENDED OFF
   END;
```

Instruct the database server to carry out an automatic backup of the transaction log only, every hour, Monday to Friday between 8 A.M. and 6 P.M.

```
CREATE EVENT HourlyLogBackup
SCHEDULE hourly_log_backup
BETWEEN '8:00AM' AND '6:00PM'
EVERY 1 HOURS ON
   ('Monday','Tuesday','Wednesday','Thursday','Friday')
HANDLER
   BEGIN
      BACKUP DATABASE DIRECTORY 'c:\\database\\backup'
      TRANSACTION LOG ONLY
      TRANSACTION LOG RENAME
   END;
```

See "Defining trigger conditions for events" [*SQL Anywhere Server - Database Administration*].

CREATE EXISTING TABLE statement

Use this statement to create a new proxy table, which represents an existing object on a remote server.

Syntax

CREATE EXISTING TABLE [*owner*.]*table-name*
[**(***column-definition*, *…***)**]
AT *location-string*

column-definition :
column-name data-type [**NOT NULL**]

location-string :
remote-server-name.[*db-name*].[*owner*].*object-name*
| *remote-server-name*;[*db-name*];[*owner*];*object-name*

Parameters

AT clause The AT clause specifies the location of the remote object. The AT clause supports the semicolon (;) as a delimiter. If a semicolon is present anywhere in the *location-string* string, the semicolon is the field delimiter. If no semicolon is present, a period is the field delimiter. This allows file names and extensions to be used in the database and owner fields. For example, the following statement maps the table a1 to the MS Access file *mydbfile.mdb*:

```
CREATE EXISTING TABLE a1
AT 'access;d:\mydbfile.mdb;;a1';
```

Remarks

The CREATE EXISTING TABLE statement creates a new local, proxy table that maps to a table at an external location. The CREATE EXISTING TABLE statement is a variant of the CREATE TABLE statement. The EXISTING keyword is used with CREATE TABLE to specify that a table already exists remotely and that its metadata is to be imported into SQL Anywhere. This establishes the remote table as a visible entity to SQL Anywhere users. SQL Anywhere verifies that the table exists at the external location before it creates the table.

If the object does not exist (either host data file or remote server object), the statement is rejected with an error message.

Index information from the host data file or remote server table is extracted and used to create rows for the ISYSIDX system table. This defines indexes and keys in server terms and enables the query optimizer to consider any indexes that may exist on this table.

Referential constraints are passed to the remote location when appropriate.

If column-definitions are not specified, SQL Anywhere derives the column list from the metadata it obtains from the remote table. If column-definitions are specified, SQL Anywhere verifies the column-definitions. Column names, data types, lengths, identity property, and null properties are checked for the following:

- Column names must match identically (although case is ignored).

- Data types in the CREATE EXISTING TABLE statement must match or be convertible to the data types of the column on the remote location. For example, a local column data type is defined as money, while the remote column data type is numeric.
- Each column's NULL property is checked. If the local column's NULL property is not identical to the remote column's NULL property, a warning message is issued, but the statement is not aborted.
- Each column's length is checked. If the length of char, varchar, binary, varbinary, decimal and numeric columns do not match, a warning message is issued, but the command is not aborted.

  You may choose to include only a subset of the actual remote column list in your CREATE EXISTING statement.

Permissions

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "CREATE TABLE statement" on page 450
- "Specifying proxy table locations" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

Create a proxy table named blurbs for the blurbs table at the remote server server_a.

```
CREATE EXISTING TABLE blurbs
( author_id ID not null,
copy text not null)
AT 'server_a.db1.joe.blurbs';
```

Create a proxy table named blurbs for the blurbs table at the remote server server_a. SQL Anywhere derives the column list from the metadata it obtains from the remote table.

```
CREATE EXISTING TABLE blurbs
AT 'server_a.db1.joe.blurbs';
```

Create a proxy table named rda_employees for the Employees table at the SQL Anywhere remote server, demo10.

```
CREATE EXISTING TABLE rda_employees
AT 'demo10...Employees';
```


CREATE EXTERNLOGIN statement

Use this statement to assign an alternate login name and password to be used when communicating with a remote server.

Syntax

CREATE EXTERNLOGIN *login-name*
TO *remote-server*
REMOTE LOGIN *remote-user*
[**IDENTIFIED BY** *remote-password*]

Parameters

login-name specifies the local user login name. When using integrated logins, the *login-name* is the database user to which the Windows user or group is mapped.

TO clause The TO clause specifies the name of the remote server.

REMOTE LOGIN clause The REMOTE LOGIN clause specifies the user account on *remote-server* for the local user *login-name*.

IDENTIFIED BY clause The IDENTIFIED BY clause specifies the *remote-password* for *remote-user*. The *remote-user* and *remote-password* combination must be valid on the remote-server.

If you omit the IDENTIFIED BY clause, the password is sent to the remote server as NULL. However, if you specify IDENTIFIED BY "" (an empty string), then the password sent is the empty string.

Remarks

By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. CREATE EXTERNLOGIN assigns an alternate login name and password to be used when communicating with a remote server.

The password is stored internally in encrypted form. The *remote-server* must be known to the local server by an entry in the ISYSSERVER table. See "CREATE SERVER statement" on page 435.

Sites with automatic password expiration should plan for periodic updates of passwords for external logins.

CREATE EXTERNLOGIN cannot be used from within a transaction.

Permissions

Only users with DBA authority can add or modify an external login for *login-name*.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "DROP EXTERNLOGIN statement" on page 502

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Map the local user named DBA to the user sa with password Plankton when connecting to the server sybase1.

```
CREATE EXTERNLOGIN DBA
TO sybase1
REMOTE LOGIN sa
IDENTIFIED BY Plankton;
```

CREATE FUNCTION statement

Use this statement to create a new function in the database.

Syntax 1

CREATE [**TEMPORARY**] **FUNCTION** [*owner*.]*function-name* **(** [*parameter*, …] **)**
RETURNS *data-type routine-characteristics*
{ *compound-statement*
  | **AS** *tsql-compound-statement*
  | *external-name* }

Syntax 2

CREATE FUNCTION [*owner*.]*function-name* **(** [*parameter*, …] **)**
RETURNS *data-type*
URL *url-string*
[**HEADER** *header-string*]
[**SOAPHEADER** *soap-header-string*]
[**TYPE** { **'HTTP**[**:**{ **GET** | **POST** }] **'** | **'SOAP**[**:**{ **RPC** | **DOC** }]**'** }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]

url-string :
'{ HTTP
  | **HTTPS**
  |**HTTPS_FIPS}:*//***[*user***:***password*@]*hostname*[**:***port*] }[***/****path*]**'**

parameter :
IN *parameter-name data-type* [**DEFAULT** *expression*]

routine-characteristics
ON EXCEPTION RESUME | [**NOT**] **DETERMINISTIC**

tsql-compound-statement:
sql-statement
sql-statement
 ...

external-name:
  EXTERNAL NAME *library-call*
| **EXTERNAL NAME** *java-call* **LANGUAGE JAVA**

library-call :
[*operating-system*:]*function-name@library*; …

operating-system :
NetWare | **Unix**

java-call :
[*package-name*.]*class-name.method-name method-signature*

method-signature :
([*field-descriptor*, …]**)** *return-descriptor*

field-descriptor | *return-descriptor* :
Z | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[***descriptor* | **L***class-name***;**

Parameters

CREATE FUNCTION clause Parameter names must conform to the rules for database identifiers. They must have a valid SQL data type, and must be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the function.

When functions are executed, not all parameters need to be specified. If a default value is provided in the CREATE FUNCTION statement, missing parameters are assigned the default values. If an argument is not provided by the caller and no default is set, an error is given.

Specifying TEMPORARY (CREATE TEMPORARY FUNCTION) means that the function is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary functions can also be explicitly dropped. You cannot perform ALTER, GRANT, or REVOKE on them, and, unlike other functions, temporary functions are not recorded in the catalog or transaction log.

Temporary functions execute with the permissions of their creator (current user), and can only be owned by their creator. Therefore, do not specify *owner* when creating a temporary function.

Temporary functions can be created and dropped when connected to a read-only database.

compound-statement A set of SQL statements bracketed by BEGIN and END, and separated by semicolons. See "BEGIN statement" on page 351

tsql-compound-statement A batch of Transact-SQL statements. See "Transact-SQL batch overview" [*SQL Anywhere Server - SQL Usage*], and "CREATE PROCEDURE statement [T-SQL]" on page 425.

EXTERNAL NAME clause A function using the EXTERNAL NAME clause is a wrapper around a call to a function in an external library. A function using EXTERNAL NAME can have no other clauses following the RETURNS clause. The *library* name may include the file extension, which is typically *.dll* on Windows, *.so* on Unix, and *.nlm* on NetWare. In the absence of the extension, the software appends the platform-specific default file extension for libraries. On NetWare, if no NLM name is given, the NLM containing the symbol must already be loaded when the function is called.

For information about external library calls, see "Calling external libraries from procedures" [*SQL Anywhere Server - SQL Usage*].

EXTERNAL NAME LANGUAGE JAVA clause A function that uses EXTERNAL NAME with a LANGUAGE JAVA clause is a wrapper around a Java method.

For information on calling Java procedures, see "CREATE PROCEDURE statement" on page 414.

ON EXCEPTION RESUME clause Use Transact-SQL -like error handling. See "CREATE PROCEDURE statement" on page 414.

NOT DETERMINISTIC clause A function specified as NOT DETERMINISTIC is re-evaluated each time it is called in a query. The results of functions not specified in this manner may be cached for better performance, and re-used each time the function is called with the same parameters during query evaluation.

Functions that have side effects such as modifying the underlying data should be declared as NOT DETERMINISTIC. For example, a function that generates primary key values and is used in an INSERT … SELECT statement should be declared NOT DETERMINISTIC:

```
CREATE FUNCTION keygen( increment INTEGER )
RETURNS INTEGER
NOT DETERMINISTIC
BEGIN
  DECLARE keyval INTEGER;
  UPDATE counter SET x = x + increment;
  SELECT counter.x INTO keyval FROM counter;
  RETURN keyval
END
INSERT INTO new_table
SELECT keygen(1), ...
FROM old_table;
```

Functions may be declared as DETERMINISTIC if they always return the same value for given input parameters. Future versions of the software may use this declaration to allow optimizations that are unsafe for functions that could return different values for the same input.

URL clause For use only when defining an HTTP or SOAP web services client function. Specifies the URL of the web service. The optional user name and password parameters provide a means of supplying the credentials needed for HTTP basic authentication. HTTP basic authentication base-64 encodes the user and password information and passes it in the "Authentication" header of the HTTP request.

Specifying HTTPS_FIPS forces the system to use the FIPS libraries. If HTTPS_FIPS is specified, but no FIPS libraries are present, non-FIPS libraries are used instead.

HEADER clause
When creating HTTP web service client functions, use this clause to add or modify HTTP request header entries. Only printable ASCII characters can be specified for HTTP headers, and they are case-insensitive. For more information about how to use this clause, see the HEADER clause of the "CREATE PROCEDURE statement" on page 414.

For more information about using HTTP headers, see "Working with HTTP headers" [*SQL Anywhere Server - Programming*].

SOAPHEADER clause When declaring a SOAP web service as a function, use this clause to specify one or more SOAP request header entries. A SOAP header can be declared as a static constant, or can be dynamically set using the parameter substitution mechanism (declaring IN, OUT, or INOUT parameters for hd1, hd2, and so on). A web service function can define one or more IN mode substitution parameters, but can not define an INOUT or OUT substitution parameter. For more information about how to use this clause, see the SOAPHEADER clause of the "CREATE PROCEDURE statement" on page 414.

For more information on using SOAP headers, see "Working with SOAP headers" [*SQL Anywhere Server - Programming*].

TYPE clause Used to specify the format used when making the web service request. If SOAP is specified or no type clause is included, the default type SOAP:RPC is used. HTTP implies HTTP:POST. Since SOAP requests are always sent as XML documents, HTTP:POST is always used to send SOAP requests.

NAMESPACE clause Applies to SOAP client functions only. This clause identifies the method namespace usually required for both SOAP:RPC and SOAP:DOC requests. The SOAP server handling the request uses this namespace to interpret the names of the entities in the SOAP request message body. The

namespace can be obtained from the WSDL description of the SOAP service available from the web service server. The default value is the procedure's URL, up to but not including the optional path component.

CERTIFICATE clause To make a secure (HTTPS) request, a client must have access to the certificate used by the HTTPS server. The necessary information is specified in a string of semicolon-separated key/value pairs. The certificate can be placed in a file and the name of the file provided using the file key, or the whole certificate can be placed in a string, but not both. The following keys are available:

Key	Abbreviation	Description
file		The file name of the certificate.
certificate	cert	The certificate itself.
company	co	The company specified in the certificate.
unit		The company unit specified in the certificate.
name		The common name specified in the certificate.

Certificates are required only for requests that are either directed to an HTTPS server, or can be redirected from a non-secure to a secure server.

CLIENTPORT clause Identifies the port number on which the HTTP client procedure communicates using TCP/IP. It is provided for and recommended only for connections across firewalls, as firewalls filter according to the TCP/UDP port. You can specify a single port number, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'.

See "ClientPort protocol option [CPORT]" [*SQL Anywhere Server - Database Administration*].

PROXY clause Specifies the URI of a proxy server. For use when the client must access the network through a proxy. Indicates that the procedure is to connect to the proxy server and send the request to the web service through it.

Remarks

The CREATE FUNCTION statement creates a user-defined function in the database. A function can be created for another user by specifying an owner name. Subject to permissions, a user-defined function can be used in exactly the same way as other non-aggregate functions.

SQL Anywhere treats all user-defined functions as deterministic unless they are declared NOT DETERMINISTIC. Deterministic functions return a consistent result for the same parameters, and are free of side effects. That is, the database server assumes that two successive calls to the same function with the same parameters will return the same result, and will not have any unwanted side-effects on the query's semantics.

If a function returns a result set, it cannot also set output parameters or return a return value.

For web service client functions, the return type of SOAP and HTTP functions must one of the character data types, such as VARCHAR. The value returned is the body of the HTTP response. No HTTP header information is included. If more information is required, such as status information, use a procedure instead of a function.

Parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the parameters are passed as part of the URL; for HTTP:POST requests, the values are placed in the body of the request. Parameters to SOAP requests are always bundled in the request body.

Permissions

Must have RESOURCE authority, unless creating a temporary function.

External functions, including Java functions, must have DBA authority.

Side effects

Automatic commit.

See also

- "ALTER FUNCTION statement" on page 310
- "CREATE PROCEDURE statement" on page 414
- "DROP statement" on page 498
- "BEGIN statement" on page 351
- "CREATE PROCEDURE statement" on page 414
- "RETURN statement" on page 634
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Examples

The following function concatenates a firstname string and a lastname string.

```
CREATE FUNCTION fullname(
   firstname CHAR(30),
   lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN (name);
END;
```

The following examples illustrate the use of the fullname function.

Return a full name from two supplied strings:

```
SELECT fullname ( 'joe', 'smith' );
```

fullname('joe', 'smith')
joe smith

List the names of all employees:

```
SELECT fullname ( GivenName, Surname )
FROM Employees;
```


fullname (GivenName, Surname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
…

The following function uses Transact-SQL syntax:

```
CREATE FUNCTION DoubleIt( @Input INT )
RETURNS INT
AS
  DECLARE @Result INT
  SELECT @Result = @Input * 2
  RETURN @Result
```

The statement `SELECT DoubleIt( 5 )` returns a value of `10`.

The following statement creates an external function written in Java:

```
CREATE FUNCTION encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
  'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA;
```


CREATE INDEX statement

Use this statement to create an index on a specified table or materialized view. Indexes can improve database performance.

Syntax 1 - Creating an index on a table

CREATE [**VIRTUAL**] [**UNIQUE**] [**CLUSTERED**] **INDEX** *index-name*
ON [*owner*.]*table-name*
(*column-name* [**ASC** | **DESC**], …
| *function-name* **(** *argument*, …] **) AS** *column-name* **)**
[{ **IN** | **ON** } *dbspace-name*]
[**FOR OLAP WORKLOAD**]

Syntax 2 - Creating an index on a materialized view

CREATE [**VIRTUAL**] [**UNIQUE**] [**CLUSTERED**] **INDEX** *index-name*
ON [*owner*.]*materialized-view-name*
(*column-name* [**ASC** | **DESC**], …**)**
[{ **IN** | **ON** } *dbspace-name*]
[**FOR OLAP WORKLOAD**]

Parameters

VIRTUAL keyword The VIRTUAL keyword is primarily for use by the Index Consultant. A virtual index mimics the properties of a real physical index during the evaluation of query plans by the Index Consultant and when the PLAN function is used. You can use virtual indexes together with the PLAN function to explore the performance impact of an index, without the often time-consuming and resource-consuming effects of creating a real index.

Virtual indexes are not visible to other connections, and are dropped when the connection is closed. Virtual indexes are not used when evaluating plans for the actual execution of queries, and so do not interfere with performance.

Virtual indexes have a limit of four columns.

See "Using the Index Consultant" [*SQL Anywhere Server - SQL Usage*], and "Index Consultant" [*SQL Anywhere Server - SQL Usage*].

CLUSTERED keyword The CLUSTERED attribute causes rows to be stored in an approximate key order corresponding to the index. While the database server makes an attempt to preserve key order, total clustering is not guaranteed.

If a clustered index exists, the LOAD TABLE statement inserts rows in the order of the index key, and the INSERT statement attempts to put new rows on the same page as the one containing adjacent rows, as defined by the key order.

See "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

UNIQUE keyword The UNIQUE attribute ensures that there will not be two rows in the table or materialized view with identical values in all the columns in the index. Each index key must be unique or contain a NULL in at least one column.

There is a difference between a unique constraint and a unique index. Columns of a unique index are allowed to be NULL, while columns in a unique constraint are not. A foreign key can reference either a primary key or a unique constraint, but not a unique index, because it can include multiple instances of NULL.

It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for primary keys or for columns with unique constraints. Approximate numeric data types are subject to rounding errors after arithmetic operations.

ASC | DESC keyword Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index is used for both an ascending and a descending ORDER BY, whether the index was ascending or descending. However, if an ORDER BY is performed with mixed ascending and descending attributes, an index is used only if the index was created with the same ascending and descending attributes.

function-name clause The function-name clause creates an index on a function. This clause cannot be used on declared temporary tables or materialized views.

This form of the CREATE INDEX statement is a convenience method that carries out the following operations:

1. Adds a computed column named *column-name* to the table. The column is defined with a COMPUTE clause that is the specified function, along with any specified arguments. See the COMPUTE clause of the CREATE TABLE statement for restrictions on the type of function that can be specified. The data type of the column is based on the result type of the function.
2. Populates the computed column for the existing rows in the table.
3. Creates an index on the column.

Dropping the index does not cause the associated computed column to be dropped.

For more information about computed columns, see "Working with computed columns" [*SQL Anywhere Server - SQL Usage*].

IN | ON clause By default, the index is placed in the same database file as its table or materialized view. You can place the index in a separate database file by specifying a dbspace name in which to put the index. This feature is useful mainly for large databases to circumvent file size limitations, or for performance improvements that might be achieved by using multiple disk devices.

For more information on limitations, see "SQL Anywhere size and number limitations" [*SQL Anywhere Server - Database Administration*].

FOR OLAP WORKLOAD option When you specify FOR OLAP WORKLOAD, the database server performs certain optimizations and gather statistics on the key to help improve performance for OLAP workloads, particularly when the optimization_workload is set to OLAP. See "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about OLAP, see "OLAP Support" [*SQL Anywhere Server - SQL Usage*].

Remarks

Syntax 1 is for use with tables; Syntax 2 is for use with materialized views.

SQL Anywhere uses physical and logical indexes. A physical index is the actual indexing structure as it is stored on disk. A logical index is a reference to a physical index. If you create an index that is identical in its physical attributes to an existing index, the database server creates a logical index that shares the existing physical index. In general, indexes created by you are considered logical indexes. The database server creates physical indexes as required to implement logical indexes, and can share the same physical index among several logical indexes. See "Index sharing using logical indexes" [*SQL Anywhere Server - SQL Usage*].

The CREATE INDEX statement creates a sorted index on the specified columns of the named table or materialized view. Indexes are automatically used to improve the performance of queries issued to the database, and to sort queries with an ORDER BY clause. Once an index is created, it is never referenced in a SQL statement again except to validate it (VALIDATE INDEX), alter it (ALTER INDEX), delete it (DROP INDEX), or in a hint to the optimizer.

- **Index ownership** There is no way of specifying the index owner in the CREATE INDEX statement. Indexes are always owned by the owner of the table or materialized view.
- **Indexes on views** You can create indexes on materialized views, but not on non-materialized views.
- **Index name space** The name of each index must be unique for a given table or materialized view.
- **Exclusive use** CREATE INDEX is prevented whenever the statement affects a table or materialized view currently being used by another connection. CREATE INDEX can be time consuming and the database server will not process requests referencing the same table while the statement is being processed.
- **Automatically created indexes** SQL Anywhere automatically creates indexes for primary key, foreign key, and unique constraints. These automatically created indexes are held in the same database file as the table.

Permissions

Must be the owner of the table or materialized view, or have either DBA authority or REFERENCES permission.

Cannot be used within a snapshot transaction. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit. Creating an index on a built-in function also causes a checkpoint.

Column statistics are updated (or created if they do not exist).

See also

- "DROP statement" on page 498
- "Indexes" [*SQL Anywhere Server - SQL Usage*]
- "CREATE STATISTICS statement" on page 442
- "Index sharing using logical indexes" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Create a two-column index on the Employees table.

```
CREATE INDEX employee_name_index
ON Employees
( Surname, GivenName );
```

Create an index on the SalesOrderItems table for the ProductID column.

```
CREATE INDEX item_prod
ON SalesOrderItems
( ProductID );
```

Use the SORTKEY function to create an index on the Description column of the Products table, sorted according to a Russian collation. As a side effect, the statement adds a computed column desc_ru to the table.

```
CREATE INDEX ix_desc_ru
ON Products (
 SORTKEY( Description, 'rusdict' )
 AS desc_ru );
```

CREATE LOCAL TEMPORARY TABLE statement

Use this statement within a procedure to create a local temporary table that persists after the procedure completes and until it is either explicitly dropped, or until the connection terminates.

Syntax

CREATE LOCAL TEMPORARY TABLE *table-name*
({ *column-definition* [*column-constraint* …] | *table-constraint* | *pctfree* }, … **)**
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS** | **NOT TRANSACTIONAL**]

pctfree : **PCTFREE** *percent-free-space*

percent-free-space : *integer*

Parameters

For definitions of *column-definition*, *column-constraint*, *table-constraint*, and *pctfree*, see "CREATE TABLE statement" on page 450.

ON COMMIT By default, the rows of a temporary table are deleted on a COMMIT. You can use the ON COMMIT clause to preserve rows on a COMMIT.

NOT TRANSACTIONAL The NOT TRANSACTIONAL clause provides performance improvements in some circumstances because operations on non-transactional temporary tables do not cause entries to be made in the rollback log. For example, NOT TRANSACTIONAL may be useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

Remarks

In a procedure, use the CREATE LOCAL TEMPORARY TABLE statement, instead of the DECLARE LOCAL TEMPORARY TABLE statement, when you want to create a table that persists after the procedure completes. Local temporary tables created using the CREATE LOCAL TEMPORARY TABLE statement remain until they are either explicitly dropped, or until the connection closes.

Local temporary tables created in IF statements using CREATE LOCAL TEMPORARY TABLE also persist after the IF statement completes.

Permissions

None.

Side effects

None.

See also

- "CREATE TABLE statement" on page 450
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 483
- "Using compound statements" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Example

The following example illustrates how to create a temporary table in a stored procedure:

```
BEGIN
  CREATE LOCAL TEMPORARY TABLE TempTab ( number INT );
  ...
END
```


CREATE MATERIALIZED VIEW statement

Use this statement to create a materialized view.

Syntax

CREATE MATERIALIZED VIEW
[*owner*.]*materialized-view-name* [**(** *column-name*, … **)**]
[**IN** *dbspace-name*]
AS *select-statement*

Parameters

column-name list Specifies the columns to create in the materialized view. If no *column-name* list is specified, the column names are set to the columns specified in the *select-statement* of the AS clause.

IN clause Specifies the dbspace in which to create the materialized view. If not specified, the current dbspace is used.

AS clause Defines the structure of the materialized view using a *select-statement*. A materialized view definition can only reference base tables. It cannot reference views, other materialized views, or temporary tables. The *select-statement* must contain column names or have an alias-name specified (see "SELECT statement" on page 648). You cannot use a SELECT * construct to specify column names. For example, you cannot specify `CREATE MATERIALIZED VIEW matview AS SELECT * FROM` *table-name* .... Also, all objects in the *select-statement* must have unique names in the database.

See "Restrictions when managing materialized views" [*SQL Anywhere Server - SQL Usage*].

Remarks

Materialized views are not automatically initialized with data when created. To initialize a materialized view, use either the REFRESH MATERIALIZED VIEW statement to initialize an individual materialized view, or the sa_refresh_materialized_views system procedure to initialize all uninitialized materialized views in the database. See "REFRESH MATERIALIZED VIEW statement" on page 621, and "sa_refresh_materialized_views system procedure" on page 907.

You can encrypt a materialized view, change its PCTFREE setting, and enable or disable its use by the optimizer. However, you must create the materialized view first, and then use the ALTER MATERIALIZED VIEW to set these options. The default values for these options at creation time are NOT ENCRYPTED, ENABLE USE IN OPTIMIZATION, and the default PCTFREE according to the page size in use for the database (200 bytes for a 4 KB page size, and 100 bytes for a 2 KB page size).

The sa_recompile_views system procedure does not attempt to recompile materialized views.

Several options need to have specific values in order to create a materialized view. See "Restrictions when managing materialized views" [*SQL Anywhere Server - SQL Usage*].

Permissions

You must have RESOURCE authority and SELECT permission on the tables in the materialized view definition. To create a materialized view for another user, you must also have DBA authority.

Side effects

While executing, the CREATE MATERIALIZED VIEW statement places exclusive locks, without blocking, on all tables referenced by the materialized view. If one of the referenced tables cannot be locked, the statement fails and an error is returned.

See also

- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "ALTER MATERIALIZED VIEW statement" on page 313
- "DROP statement" on page 498
- "REFRESH MATERIALIZED VIEW statement" on page 621
- "CREATE VIEW statement" on page 471

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a materialized view containing confidential information about employees in the SQL Anywhere sample database. You must subsequently execute a REFRESH MATERIALIZED VIEW statement, to initialize the view for use.

```
CREATE MATERIALIZED VIEW EmployeeConfidential AS
SELECT EmployeeID, Employees.DepartmentID,
    SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
FROM Employees, Departments
WHERE Employees.DepartmentID=Departments.DepartmentID
ORDER BY Employees.DepartmentID;
```


CREATE MESSAGE statement [T-SQL]

Use this statement to add a user-defined message to the ISYSUSERMESSAGE system table for use by PRINT and RAISERROR statements.

Syntax

CREATE MESSAGE *message-number* **AS** *message-text*

message-number : *integer*

message-text : *string*

Parameters

message_number The message number of the message to add. The message number for a user-defined message must be 20000 or greater.

message_text The text of the message to add. The maximum length is 255 bytes. PRINT and RAISERROR recognize placeholders in the message text. A single message can contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow the message when the text of the message is sent to the client.

The placeholders are numbered to allow reordering of the arguments when translating a message to a language with a different grammatical structure. A placeholder for an argument appears as "%nn!": a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation mark (!), where the integer represents the position of the argument in the argument list. "%1!" is the first argument, "%2!" is the second argument, and so on.

There is no parameter corresponding to the *language* argument for **sp_addmessage**.

Remarks

CREATE MESSAGE associates a message number with a message string. The message number can be used in PRINT and RAISERROR statements.

To drop a message, see "DROP statement" on page 498.

Permissions

Must have RESOURCE authority

Side effects

Automatic commit.

See also

- "PRINT statement [T-SQL]" on page 613
- "RAISERROR statement [T-SQL]" on page 616
- "ISYSUSERMESSAGE system table" on page 734

Standards and compatibility

- **SQL/2003** Vendor extension.

CREATE PROCEDURE statement

Use this statement to create a procedure in the database.

Syntax 1 - Creating user defined procedures

CREATE [**TEMPORARY**] **PROCEDURE** [*owner*.]*procedure-name* **(** [*parameter*, …] **)**
{ [**RESULT (** *result-column*, … **)** | **NO RESULT SET**]
[**ON EXCEPTION RESUME**]
compound-statement
| **AT** *location-string*
| **EXTERNAL NAME** *library-call*
| [**DYNAMIC RESULT SETS** *integer-expression*]
[**EXTERNAL NAME** *java-call* **LANGUAGE JAVA**]
}

parameter :
parameter-mode parameter-name data-type [**DEFAULT** *expression*]
| **SQLCODE**
| **SQLSTATE**

parameter-mode : **IN** | **OUT** | **INOUT**

result-column : *column-name data-type*

library-call :
[*operating-system*:]*function-name@library*; …

operating-system : **NetWare** | **Unix**

java-call :
[*package-name*.]*class-name.method-name method-signature*

method-signature :
([*field-descriptor*, …] **)** *return-descriptor*

field-descriptor | *return-descriptor* :
Z | **B** | **S** | **I** | **J** | **F** | **D** | **C** | **V** | **[***descriptor* | **L***class-name***;**

Syntax 2 - Create web services

CREATE PROCEDURE [*owner*.]*procedure-name* **(** [*parameter*, …] **)**
URL *url-string*
[**HEADER** *header-string*]
[**SOAPHEADER** *soap-header-string*]
[**TYPE** { **'HTTP**[**:**{ **GET** | **POST** }]' | **'SOAP**[**:**{ **RPC** | **DOC** }]' }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]
[**SET** *protocol-option-string*

parameter :
parameter-mode parameter-name data-type [**DEFAULT** *expression*]
| **SQLCODE**
| **SQLSTATE**

parameter-mode : **IN** | **OUT** | **INOUT**

url-string :
{ **HTTP** | **HTTPS** | **HTTPS_FIPS** }**://**[*user***:***password*@]*hostname*[**:***port*][**/***path*]

header-string :
The string to use for the HTTP header.

protocol-option-string
[*http-option-list*]
[**,** *soap-option-list*]

http-option-list :
HTTP (
[**CH**[**UNK**]={ **ON** | **OFF** | **AUTO** }]
[**; VER**[**SION**]={ **1.0** | **1.1** }]
)

soap-option-list:
SOAP (
OP[**ERATION**]=*soap-operation-name*
)

soap-operation-name :
The name of the SOAP operation to call.

Parameters

CREATE PROCEDURE clause You can create permanent or temporary (TEMPORARY) stored procedures. You can use PROC as a synonym for PROCEDURE.

Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type (see "SQL Data Types" on page 47). Parameters can be prefixed with one of the keywords IN, OUT, or INOUT. If you do not specify one of these values, parameters are INOUT by default. The keywords have the following meanings:

- **IN** The parameter is an expression that provides a value to the procedure.
- **OUT** The parameter is a variable that could be given a value by the procedure.
- **INOUT** The parameter is a variable that provides a value to the procedure, and could be given a new value by the procedure.

When procedures are executed using the CALL statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If an argument is not provided in the CALL statement, and no default is set, an error is given.

SQLSTATE and SQLCODE are special parameters that output the SQLSTATE or SQLCODE value when the procedure ends (they are OUT parameters). Whether or not a SQLSTATE and SQLCODE parameter is specified, the SQLSTATE and SQLCODE special values can always be checked immediately after a procedure call to test the return status of the procedure.

The SQLSTATE and SQLCODE special values are modified by the next SQL statement. Providing SQLSTATE or SQLCODE as procedure arguments allows the return code to be stored in a variable.

Specifying TEMPORARY (CREATE TEMPORARY PROCEDURE) means that the stored procedure is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary stored procedures can also be explicitly dropped. You cannot perform ALTER, GRANT, or REVOKE on them, and, unlike other stored procedures, temporary stored procedures are not recorded in the catalog or transaction log.

Temporary stored procedures execute with the permissions of their creator (current user), and can only be owned by their creator. Therefore, do not specify *owner* when creating a temporary stored procedure.

Temporary stored procedures can be created and dropped when connected to a read-only database, and they cannot be external procedures.

For example, the following temporary procedure drops the table called CustRank, if it exists. For this example, the procedure assumes that the table name is unique and can be referenced by the procedure creator without specifying the table owner:

```
CREATE TEMPORARY PROCEDURE drop_table( IN CustRank char(128) )
BEGIN
    IF EXISTS ( SELECT * FROM SYS.SYSTAB WHERE table_name = CustRank ) THEN
   EXECUTE IMMEDIATE 'DROP TABLE "' || CustRank || '"';
   MESSAGE 'Table "' || CustRank || '" dropped' to client;
    END IF;
END;
```

RESULT clause The RESULT clause declares the number and type of columns in the result set. The parenthesized list following the RESULT keyword defines the result column names and types. This information is returned by the embedded SQL DESCRIBE or by ODBC SQLDescribeCol when a CALL statement is being described. Allowable data types are listed in "SQL Data Types" on page 47.

For more information on returning result sets from procedures, see "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*].

Some procedures can produce more than one result set, with different numbers of columns, depending on how they are executed. For example, the following procedure returns two columns under some circumstances, and one in others.

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
   IF formal = 'n' THEN
      SELECT GivenName
      FROM Employees
   ELSE
      SELECT Surname, GivenName
      FROM Employees
   END IF
END;
```

Procedures with variable result sets must be written without a RESULT clause, or in Transact-SQL. Their use is subject to the following limitations:

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, to get the proper shape of result set. The CURSOR *cursor-name* clause on the DESCRIBE statement is required.

- **ODBC, OLE DB, ADO.NET** Variable result-set procedures can be used by applications using these interfaces. The proper description of the result sets is carried out by the driver or provider.
- **Open Client applications** Variable result-set procedures can be used by Open Client applications.

If your procedure returns only one result set, you should use a RESULT clause. The presence of this clause prevents ODBC and Open Client applications from re-describing the result set after a cursor is open.

To handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure's defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the procedure definition. To avoid this problem, use column aliases in the SELECT statement that generates the result set.

NO RESULT SET clause Declares that no result set is returned by this procedure. This is useful when an external environment needs to know that a procedure does not return a result set.

ON EXCEPTION RESUME clause This clause enables Transact-SQL -like error handling to be used within a Watcom-SQL syntax procedure.

If you use ON EXCEPTION RESUME, the procedure takes an action that depends on the setting of the on_tsql_error option. If on_tsql_error is set to Conditional (which is the default) the execution continues if the next statement handles the error; otherwise, it exits.

Error-handling statements include the following:

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

You should not use explicit error handling code with an ON EXCEPTION RESUME clause.

See “on_tsql_error option [compatibility]” [*SQL Anywhere Server - Database Administration*].

AT location-string clause Create a proxy stored procedure on the current database for a remote procedure specified by *location-string*. The AT clause supports the semicolon (;) as a field delimiter in *location-string*. If no semicolon is present, a period is the field delimiter. This allows file names and extensions to be used in the database and owner fields.

Remote procedures can return only up to 254 characters in output variables.

For information on remote servers, see “CREATE SERVER statement” on page 435. For information on using remote procedures, see “Using remote procedure calls (RPCs)” [*SQL Anywhere Server - SQL Usage*].

EXTERNAL NAME clause A procedure using the EXTERNAL NAME clause is a wrapper around a call to an external library. A stored procedure using EXTERNAL NAME can have no other clauses following the parameter list. The *library* name may include the file extension, which is typically *.dll* on Windows, *.so* on Unix, and *.nlm* on NetWare. In the absence of the extension, the software appends the platform-specific default file extension for libraries. On NetWare, if no NLM name is given, the NLM containing the symbol must already be loaded when the function is called.

For information about external library calls, see "Calling external libraries from procedures" [*SQL Anywhere Server - SQL Usage*].

DYNAMIC RESULT SETS clause This clause is directly tied to the EXTERNAL NAME LANGUAGE JAVA clause, and is for use with procedures that are wrappers around Java methods. If the DYNAMIC RESULT SETS clause is not provided, it is assumed that the method returns no result set.

EXTERNAL NAME java-call LANGUAGE JAVA clause A procedure that uses EXTERNAL NAME with a LANGUAGE JAVA clause is a wrapper around a Java method. A Java method signature is a compact character representation of the types of the parameters and the type of the return value. If the number of parameters is less than the number indicated in the method-signature then the difference must equal the number specified in DYNAMIC RESULT SETS, and each parameter in the method signature in excess of those in the procedure parameter list must have a method signature of `[Ljava/SQL/ResultSet;`.

The *field-descriptor* and *return-descriptor* have the following meanings:

Field type	Java data type
B	byte
C	char
D	double
F	float
I	int
J	long
L *class-name*;	an instance of the class *class-name*. The class name must be fully qualified, and any dot in the name must be replaced by a /. For example, java/lang/String
S	short
V	void
Z	Boolean
[	use one for each dimension of an array

For example,

```
double some_method(
  boolean a,
  int b,
```



```
    java.math.BigDecimal c,
    byte [][] d,
    java.sql.ResultSet[] rs ) {
}
```

would have the following signature:

```
'(ZILjava/math/BigDecimal;[[B[Ljava/SQL/ResultSet;)D'
```

See "Returning result sets from Java methods" [*SQL Anywhere Server - Programming*].

URL clause For use only when defining an HTTP or SOAP web services client procedure. Specifies the URI of the web service. The optional user name and password parameters provide a means of supplying the credentials needed for HTTP basic authentication. HTTP basic authentication base-64 encodes the user and password information and passes it in the Authentication header of the HTTP request.

Specifying HTTPS_FIPS forces the system to use the FIPS libraries. If HTTPS_FIPS is specified, but no FIPS libraries are present, non-FIPS libraries are used instead.

When specified in this way, the user name and password are passed unencrypted, as part of the URL.

HEADER clause

When creating HTTP web service client procedures, use this clause to modify HTTP request header entries, add new ones, or suppress existing headers. The specification of headers closely resembles the format specified in RFC2616 Hypertext Transfer Protocol — HTTP/1.1, and RFC822 Standard for ARPA Internet Text Messages, including the fact that only printable ASCII characters can be specified for HTTP headers, and they are case-insensitive. Following are a few key points regarding HTTP header specification:

- Header/value pairs can be delimited by \n or \x0d\n, specifying Line Feed (<LF>), or Carriage Return and Line Feed(<CR><LF>), respectively.
- A header is delimited from its value using a colon (:), and therefore cannot contain a colon.
- A header followed by :\n, or an end of line, specifies a header with no value. Similarly, a header with no colon or value after. For example, `HEADER 'Date'`, specifies that the Date header not be included. Suppressing headers, or their values, can cause unexpected results. See "Modifying HTTP headers" [*SQL Anywhere Server - Programming*].
- Folding of long header values is supported, provided one or more white spaces immediately follow the \n. For example, the following HEADER specification, and resulting HTTP output, are semantically equivalent:

  ```
  ... HEADER 'heading1: This long value\n is a really long value for heading1
  \n
  heading2:shortvalue'

  heading1:This long value is a really long value for heading1<CR><LF>
  heading2:shortvalue<CR><LF>
  ```

- Multiple contiguous white spaces, including folding, results in a single white space.
- Parameter substitution is supported for this clause.

This example shows how to add static user-defined headers:


```
CREATE PROCEDURE http_client() URL 'http://localhost/getdata'
  TYPE 'http:get' HEADER 'UserHeader1:value1\nUserHeader2:value2';
```

This example shows how to add new parameter-substituted user-defined headers:

```
CREATE PROCEDURE http_client( headers LONG VARCHAR ) URL 'http://localhost/
getdata'
TYPE 'http:get' HEADER '!headers';
CALL http_client( 'NewHeader1:value1\nNewHeader2:value2' );
```

For more information about using HTTP headers, see "Working with HTTP headers" [*SQL Anywhere Server - Programming*].

SOAPHEADER clause When declaring a SOAP web service as a procedure, use this clause to specify one or more SOAP request header entries. A SOAP header can be declared as a static constant, or can be dynamically set using the parameter substitution mechanism (declaring IN, OUT, or INOUT parameters for hd1, hd2, and so on). A web service procedure can define one or more IN mode substitution parameters, and a single INOUT or OUT substitution parameter.

The following example illustrates how a client can specify the sending of several header entries with parameters and receiving the response SOAP header data:

```
CREATE PROCEDURE soap_client( INOUT VARCHAR hd1, IN VARCHAR hd2, IN VARCHAR
hd3 )
  URL 'localhost/some_endpoint'
  SOAPHEADER '!hd1!hd2!hd3';
```

For more information on using SOAP headers, see "Working with SOAP headers" [*SQL Anywhere Server - Programming*].

TYPE clause Used to specify the format used when making the web service request. If SOAP is specified or no type clause is included, the default type SOAP:RPC is used. HTTP implies HTTP:POST. Since SOAP requests are always sent as XML documents, HTTP:POST is always used to send SOAP requests.

NAMESPACE clause Applies to SOAP client procedures only. This clause identifies the method namespace usually required for both SOAP:RPC and SOAP:DOC requests. The SOAP server handling the request uses this namespace to interpret the names of the entities in the SOAP request message body. The namespace can be obtained from the WSDL description of the SOAP service available from the web service server. The default value is the procedure's URL, up to but not including the optional path component.

CERTIFICATE clause To make a secure (HTTPS) request, a client must have access to the certificate used by the HTTPS server. The necessary information is specified in a string of semicolon-separated key/value pairs. The certificate can be placed in a file and the name of the file provided using the file key, or the whole certificate can be placed in a string, but not both. The following keys are available:

Key	Abbreviation	Description
file		The file name of the certificate.
certificate	cert	The certificate itself.
company	co	The company specified in the certificate.

Key	Abbrevia-tion	Description
unit		The company unit specified in the certificate.
name		The common name specified in the certificate.

Certificates are required only for requests that are either directed to an HTTPS server, or can be redirected from a non-secure to a secure server.

CLIENTPORT clause Identifies the port number on which the HTTP client procedure communicates using TCP/IP. It is provided for and recommended only for connections across firewalls, as firewalls filter according to the TCP/UDP port. You can specify a single port number, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'.

See "ClientPort protocol option [CPORT]" [*SQL Anywhere Server - Database Administration*].

PROXY clause Specifies the URI of a proxy server. For use when the client must access the network through a proxy. Indicates that the procedure is to connect to the proxy server and send the request to the web service through it.

SET clause Specifies protocol-specific behavior options for HTTP and SOAP. The following list describes the supported SET options. CHUNK and VERSION apply to the HTTP protocol, and OPERATION applies to the SOAP protocol. Parameter substitution is supported for this clause.

- **CH or CHUNK** This option allows you to specify whether to use chunking. Chunking allows HTTP messages to be broken up into several parts. Possible values are ON (always chunk), OFF (never chunk), and AUTO (chunk only if the contents, excluding auto-generated markup, exceeds 2048 bytes). For example, the following SET clause enables chunking:

  ```
  .. SET 'HTTP ( CHUNK=ON )' ..
  ```

  If the CHUNK option is not specified, the default behavior is AUTO. If chunking fails in AUTO mode with a status of 505 ('HTTP Version Not Supported'), or with 501 ('Not Implemented'), client retries the request without chunked encoding.

  Since CHUNK mode is a transfer encoding supported starting in HTTP version 1.1, setting CHUNK to ON requires that the version (VER) be set to 1.1, or not be set at all, in which case 1.1 is used as the default version.

- **VER or VERSION** This option allows you to specify the version of HTTP protocol that is used for the format of the HTTP message. For example, the following SET clause sets the HTTP version to 1.1:

  ```
  ... SET 'HTTP ( VERSION=1.1 )' ...
  ```

  Possible values are 1.0 and 1.1. If VERSION is not specified:

  - if CHUNK is set to ON, 1.1 is used as the HTTP version
  - if CHUNK is set to OFF, 1.0 is used as the HTTP version
  - if CHUNK is set to AUTO, either 1.0 or 1.1 is used, depending on whether the client is sending in CHUNK mode

- **OP or OPERATION** This option allows you to specify the name of the SOAP operation, if it is different from the name of the procedure you are creating. The value of OPERATION is analogous to the name of a remote procedure call. For example, if you wanted to create a procedure called accounts_login that calls a SOAP operation called login, you would specify something like the following:

```
CREATE PROCEDURE accounts_login(
      name LONG VARCHAR,
      pwd LONG VARCHAR )
   SET 'SOAP ( OPERATION=login )'
   ...
```

  If the OPERATION option is not specified, the name of the SOAP operation must match the name of the procedure you are creating.

The following statement shows how several *protocol-option* settings are combined in the same SET clause:

```
CREATE PROCEDURE accounts_login(
      name LONG VARCHAR,
      pwd LONG VARCHAR )
   SET 'HTTP ( CHUNK=ON; VERSION=1.1 ), SOAP( OPERATION=login )'
   ...
```

For more information on creating web services, including examples, see "SQL Anywhere Web Services" [*SQL Anywhere Server - Programming*].

Remarks

The CREATE PROCEDURE statement creates a procedure in the database. Users with DBA authority can create procedures for other users by specifying an owner. A procedure is invoked with a CALL statement.

If a stored procedure returns a result set, it cannot also set output parameters or return a return value.

For web service client procedures, parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the parameters are passed as part of the URL; for HTTP:POST requests, the values are placed in the body of the request. Parameters to SOAP requests are always bundled in the request body.

Permissions

Must have RESOURCE authority, unless creating a temporary procedure.

Must have DBA authority for external procedures or to create a procedure for another user.

Side effects

Automatic commit.

See also

- "BEGIN statement" on page 351
- "CALL statement" on page 357
- "CREATE FUNCTION statement" on page 399
- "CREATE PROCEDURE statement [T-SQL]" on page 425
- "ALTER PROCEDURE statement" on page 315
- "DROP statement" on page 498
- "EXECUTE IMMEDIATE statement [SP]" on page 519

- "GRANT statement" on page 548
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. The syntax extensions for Java result sets are as specified in the optional J621 feature.

Examples

The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType (IN product_ID INT, OUT type CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20);
   SELECT name INTO prod_name FROM Products
   WHERE ID = product_ID;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
   WHEN 'Shorts' THEN
      SET type = 'Shorts'
   ELSE
      SET type = 'UNKNOWN'
   END CASE;
END;
```

The following procedure uses a cursor and loops over the rows of the cursor to return a single value.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
      SELECT CompanyName,
          CAST(SUM(SalesOrderItems.Quantity *
          Products.UnitPrice) AS INTEGER) VALUE
      FROM Customers
      LEFT OUTER JOIN SalesOrders
      LEFT OUTER JOIN SalesOrderItems
      LEFT OUTER JOIN Products
      GROUP BY CompanyName;
   DECLARE ThisValue INT;
   DECLARE ThisCompany CHAR(35);
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop;
      END IF;
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue;
         SET TopCompany = ThisCompany;
```



```
            END IF;
        END LOOP CustomerLoop;
        CLOSE curThisCust;
    END;
```

CREATE PROCEDURE statement [T-SQL]

Use this statement to create a new procedure in the database in a manner compatible with Adaptive Server Enterprise.

Syntax 1

The following subset of the Transact-SQL CREATE PROCEDURE statement is supported in SQL Anywhere.

CREATE PROCEDURE [*owner*.]*procedure_name*
[**NO RESULT SET**]
[[**(**] *@parameter_name data-type* [= *default*] [**OUTPUT**], … [**)**]]
[**WITH RECOMPILE**] **AS** *statement-list*

Parameters

NO RESULT SET clause Declares that no result set is returned by this procedure. This is useful when an external environment needs to know that a procedure does not return a result set.

Remarks

The following differences between Transact-SQL and SQL Anywhere statements (Watcom-SQL) are listed to help those writing in both dialects.

- **Variable names prefixed by @** The "@" sign denotes a Transact-SQL variable name, while Watcom-SQL variables can be any valid identifier, and the @ prefix is optional.
- **Input and output parameters** Watcom-SQL procedure parameters are INOUT by default or can specified as IN, OUT, or INOUT. Transact-SQL procedure parameters are INPUT parameters by default or can be specified as OUTPUT. Those parameters that would be declared as INOUT or as OUT in SQL Anywhere should be declared with OUTPUT in Transact-SQL.
- **Parameter default values** Watcom-SQL procedure parameters are given a default value using the keyword DEFAULT, while Transact-SQL uses an equality sign (=) to provide the default value.
- **Returning result sets** Watcom-SQL uses a RESULT clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

  The following Transact-SQL procedure illustrates how result sets are returned from Transact-SQL stored procedures:

```
CREATE PROCEDURE showdept @deptname varchar(30)
AS
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = @deptname
   AND Departments.DepartmentID = Employees.DepartmentID;
```

  The following is the corresponding Watcom-SQL procedure:

```
CREATE PROCEDURE showdept(in deptname
         varchar(30) )
```



```
RESULT ( lastname char(20), firstname char(20))
ON EXCEPTION RESUME
BEGIN
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = deptname
   AND Departments.DepartmentID = Employees.DepartmentID
END;
```

- **Procedure body** The body of a Transact-SQL procedure is a list of Transact-SQL statements prefixed by the AS keyword. The body of a Watcom-SQL procedure is a compound statement, bracketed by BEGIN and END keywords.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "CREATE PROCEDURE statement" on page 414

Standards and compatibility

- **SQL/2003** Transact-SQL extension.
- **Sybase** SQL Anywhere supports a subset of the Adaptive Server Enterprise CREATE PROCEDURE statement syntax.

  If the Transact-SQL WITH RECOMPILE optional clause is supplied, it is ignored. SQL Anywhere always recompiles procedures the first time they are executed after a database is started, and stores the compiled procedure until the database is stopped.

  Groups of procedures are not supported.

CREATE PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to create a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax 1 (MobiLink general use)

CREATE PUBLICATION [*owner*.]*publication-name*
(*article-definition*, … **)**

article-definition :
  TABLE *table-name* [**(** *column-name*, … **)**]
[**WHERE** *search-condition*]

Syntax 2 (MobiLink scripted upload)

CREATE PUBLICATION [*owner*.]*publication-name*
WITH SCRIPTED UPLOAD
(*article-definition*, … **)**

article-definition :
  TABLE *table-name* [**(** *column-name*, … **)**]
[**USING (** [**PROCEDURE**] [*owner***.**][*procedure-name*]
  FOR UPLOAD { **INSERT** | **DELETE** | **UPDATE** }, ... **)**]

Syntax 3 (MobiLink download-only publications)

CREATE PUBLICATION [*owner*.]*publication-name*
FOR DOWNLOAD ONLY
(*article-definition*, … **)**

article-definition : **TABLE** *table-name* [**(** *column-name*, … **)**]

Syntax 4 (SQL Remote)

CREATE PUBLICATION [*owner*.]*publication-name*
(*article-definition*, … **)**

article-definition :
  TABLE *table-name* [**(** *column-name*, … **)**]
[**WHERE** *search-condition*]
[**SUBSCRIBE BY** *expression*]

Parameters

article-definition Publications are built from articles. To include more than one article, separate article definitions with commas. Each article is a table or part of a table. An article can be a vertical partition of a table (a subset of the table's columns), a horizontal partition (a subset of the table's rows based on a WHERE clause) or a vertical and horizontal partition.

In Syntax 2, which is used for publications that perform scripted uploads, the article description also registers the scripts that are used to define the upload. See "Creating publications for scripted upload" [*MobiLink - Client Administration*].

In Syntax 3, which is used for download-only publications, the article specifies only the tables and columns to be downloaded.

WHERE clause The WHERE clause is a way of defining the subset of rows of a table to be included in an article.

In MobiLink applications, the WHERE clause affects the rows included in the upload. (The download is defined by the download_cursor script.) In MobiLink SQL Anywhere remote databases, the WHERE clause can only refer to columns included in the article, and cannot contain subqueries, variables, or non-deterministic functions.

SUBSCRIBE BY clause In SQL Remote, one way of defining a subset of rows of a table to be included in an article is to use a SUBSCRIBE BY clause. This clause allows many different subscribers to receive different rows from a table in a single publication definition.

Remarks

The CREATE PUBLICATION statement creates a publication in the database. A publication can be created for another user by specifying an owner name.

In MobiLink, publications are required in SQL Anywhere remote databases, and are optional in UltraLite databases. These publications and the subscriptions to them determine which data is uploaded to the MobiLink server.

You set options for a MobiLink publication with the ADD OPTION clause in the CREATE SYNCHRONIZATION SUBSCRIPTION statement or ALTER SYNCHRONIZATION SUBSCRIPTION statement.

Syntax 2 creates a publication for scripted uploads. Use the USING clause to register the stored procedures that you want to use to define the upload. For each table, you can use up to three stored procedures: one each for inserts, deletes, and updates.

Syntax 3 creates a download-only publication that can be synchronized with no log file. When download-only publications are synchronized, downloaded rows may overwrite changes that were made to those rows in the remote database.

In SQL Remote, publishing is a two-way operation, as data can be entered at both consolidated and remote databases. In a SQL Remote installation, any consolidated database and all remote databases must have the same publication defined. Running the SQL Remote extraction utility from a consolidated database automatically executes the correct CREATE PUBLICATION statement in the remote database.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the statement.

Side effects

Automatic commit.

See also

- “ALTER PUBLICATION statement [MobiLink] [SQL Remote]” on page 317
- “DROP PUBLICATION statement [MobiLink] [SQL Remote]” on page 503
- “CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]” on page 445

- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 328
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "UltraLite CREATE PUBLICATION statement" [*UltraLite - Database Management and Reference*]
- SQL Remote: "Publishing data" [*SQL Remote*]
- "Scripted Upload" [*MobiLink - Client Administration*]
- "Download-only publications" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement publishes all columns and rows of two tables.

```
CREATE PUBLICATION pub_contact (
   TABLE Contacts,
   TABLE Company
);
```

The following statement publishes only some columns of one table.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City )
);
```

The following statement publishes only the active customer rows by including a WHERE clause that tests the Status column of the Customers table.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City, State, Status )
   WHERE Status = 'active'
);
```

The following statement publishes only some rows by providing a subscribe-by value. This method can be used only with SQL Remote.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City, State )
   SUBSCRIBE BY State
);
```

The subscribe-by value is used as follows when you create a SQL Remote subscription.

```
CREATE SUBSCRIPTION TO pub_customer ( 'NY' )
   FOR jsmith;
```

The following example creates a MobiLink publication that uses scripted uploads:

```
CREATE PUBLICATION pub WITH SCRIPTED UPLOAD (
      TABLE t1 (a, b, c) USING (
         PROCEDURE my.t1_ui FOR UPLOAD INSERT,
         PROCEDURE my.t1_ud FOR UPLOAD DELETE,
         PROCEDURE my.t1_uu FOR UPLOAD UPDATE
      ),
      TABLE t2 AS my_t2 USING (
         PROCEDURE my.t2_ui FOR UPLOAD INSERT
```



```
        )
    );
```

The following example creates a download-only publication:

```
CREATE PUBLICATION p1 FOR DOWNLOAD ONLY (
   TABLE t1
);
```


CREATE REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to identify a message-link and return address for outgoing messages from a database.

Syntax

CREATE REMOTE MESSAGE TYPE *message-system*
ADDRESS *address*

message-system: **FILE** | **FTP** | **MAPI** | **SMTP** | **VIM**

address: *string*

Parameters

message-system One of the supported message systems.

address The address for the specified message system.

Remarks

The Message Agent sends outgoing messages from a database using one of the supported message links. Return messages for users employing the specified link are sent to the specified address as long as the remote database is created by the extraction utility. The Message Agent starts links only if it has remote users for those links.

The address is the publisher's address under the specified message system. If it is an email system, the address string must be a valid email address. If it is a file-sharing system, the address string is a subdirectory of the directory set in the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

The Initialization utility creates message types automatically, without an address. Unlike other CREATE statements, the CREATE REMOTE MESSAGE TYPE statement does not give an error if the type exists; instead it alters the type.

Note
Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 555
- "GRANT REMOTE statement [SQL Remote]" on page 556
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 553
- "DROP REMOTE MESSAGE TYPE statement [SQL Remote]" on page 504

- "Using message types" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When remote databases are extracted using the extraction utility, the following statement sets all recipients of file message-system messages to send messages back to the *company* subdirectory.

The statement also instructs dbremote to look in the *company* subdirectory for incoming messages.

```
CREATE REMOTE MESSAGE TYPE file
ADDRESS 'company';
```


CREATE SCHEMA statement

Use this statement to create a collection of tables, views, and permissions for a database user.

Syntax

CREATE SCHEMA AUTHORIZATION *userid*
[
create-table-statement
| *create-view-statement*
| *grant-statement*
] ... **;**

Remarks

The CREATE SCHEMA statement creates a schema. A schema is a collection of tables, views, and their associated permissions.

The *userid* must be the user ID of the current connection. You cannot create a schema for another user.

If any statement contained in the CREATE SCHEMA statement fails, the entire CREATE SCHEMA statement is rolled back.

The CREATE SCHEMA statement is simply a way of collecting together individual CREATE and GRANT statements into one operation. There is no SCHEMA database object created in the database, and to drop the objects you must use individual DROP TABLE or DROP VIEW statements. To revoke permissions, you must use a REVOKE statement for each permission granted.

The individual CREATE or GRANT statements are not separated by statement delimiters. The statement delimiter marks the end of the CREATE SCHEMA statement itself.

The individual CREATE or GRANT statements must be ordered such that the objects are created before permissions are granted on them.

Although you can currently create more than one schema for a user, this is not recommended, and may not be supported in future releases.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- “CREATE TABLE statement” on page 450
- “CREATE VIEW statement” on page 471
- “GRANT statement” on page 548

Standards and compatibility

- **SQL/2003** Core feature.

- **Sybase** SQL Anywhere does not support the use of REVOKE statements within the CREATE SCHEMA statement, and does not allow its use within Transact-SQL batches or procedures.

Example

The following CREATE SCHEMA statement creates a schema consisting of two tables. The statement must be executed by the user ID sample_user, who must have RESOURCE authority. If the statement creating table t2 fails, neither table is created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY )
CREATE TABLE t2 ( id2 INT PRIMARY KEY );
```

The statement delimiter in the following CREATE SCHEMA statement is placed after the first CREATE TABLE statement. As the statement delimiter marks the end of the CREATE SCHEMA statement, the example is interpreted as a two statement batch by the database server. Consequently, if the statement creating table t2 fails, the table t1 is still created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY );
CREATE TABLE t2 ( id2 INT PRIMARY KEY );
```


CREATE SERVER statement

Use this statement to create a remote server.

Syntax 1

CREATE SERVER *server-name*
CLASS *server-class*
USING *connection-info*
[**READ ONLY**]

server-class :
SAJDBC | **ASEJDBC** | **SAODBC** | **ASEODBC**
| **DB2ODBC** | **MSSODBC** | **ORAODBC** | **ODBC**

connection-info :
{ *computer-name***:***port-number* [*/dbname*] | *data-source-name* | *sqlanywhere-connection-string* }

Syntax 2

CREATE SERVER *server-name*
CLASS '**DIRECTORY**'
USING *using-clause*

using-clause :
ROOT=*path*
[**;SUBDIRS**=*n*]
[**;READONLY**={ **YES** | **NO** }]

Parameters

CLASS clause Specifies the server class you want to use for a remote connection. Server classes contain detailed server capability information. If you are using NetWare, only the SAJDBC class is supported. The DIRECTORY class is used in Syntax 2 to access a directory on the local computer.

USING clause In Syntax 1, the USING clause supplies a connection string for the database server. The appropriate connection string depends on the driver being used, which in turn depends on the *server-class*.

If a JDBC-based server class is used, the USING clause is of the form *hostname*:*portnumber* [*/dbname*], where:

- **hostname** The computer the remote server runs on.
- **portnumber** The TCP/IP port number the remote server listens on. The default port number for SQL Anywhere is 2638.
- **dbname** For SQL Anywhere remote servers, if you do not specify a *dbname*, then the default database is used. For Adaptive Server Enterprise, the default is the **master** database, and an alternative to using *dbname* is to another database by some other means (for example, in the FORWARD TO statement).

If an ODBC-based server class is used, the USING clause is the *data-source-name*. The *data-source-name* is the ODBC Data Source Name.

For SQL Anywhere remote servers (SAJDBC or SAODBC server classes), the *connection-info* parameter can be any valid SQL Anywhere connection string. You can use any SQL Anywhere connection parameters.

For example, if you have connection problems, you can include a LOG connection parameter to troubleshoot the connection attempt.

For more information on SQL Anywhere connection strings, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

On Unix platforms, you need to reference the ODBC driver manager as well. For example, using the supplied iAnywhere Solutions ODBC drivers, the syntax is as follows:

```
USING 'driver=SQL Anywhere 10;dsn=my_dsn'
```

In Syntax 2, the USING clause specifies the following values for the local directory:

- **ROOT** The path, relative to the database server, that is the root of the directory access class. When you create a proxy table using the directory access server name, the proxy table is relative to this root path.
- **SUBDIRS** A number between 0 and 10 that represents the number of levels of directories within the root that the database server can access. If SUBDIRS is omitted or set to 0, then only the files in the root directory are accessible via the directory access server. You can create proxy tables to any of the directories or subdirectories available via the directory access server.
- **READONLY** Specifies whether the files accessed by the directory access server can be modified. By default, this is set to NO.

Remarks

When you create a remote sever, it is added to the ISYSSERVER system table.

Syntax 1 The CREATE SERVER statement defines a remote server.

For more information on server classes and how to configure a server, see "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 The CREATE SERVER statement lets you create a directory access server that accesses the local directory structure on the computer where the database server is running. You must create an external login for each database user that needs to use the directory access server. On Unix, the database server runs as a specific user, so file permissions are based on the permissions granted to the database server user.

For more information about directory access servers, see "Using directory access servers" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have DBA authority to execute this command.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "ALTER SERVER statement" on page 321
- "DROP SERVER statement" on page 505

- "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*]
- "ISYSSERVER system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a SQL Anywhere remote server named testsa, located on the computer named apple and listening on port number 2638, use:

```
CREATE SERVER testsa
CLASS 'SAJDBC'
USING 'apple:2638';
```

The following example creates a remote server for the JDBC-based Adaptive Server named ase_prod. Its computer name is banana and its port number is 3025.

```
CREATE SERVER ase_prod
CLASS 'asejdbc'
USING 'banana:3025';
```

The following example creates a remote server for the Oracle server named oracle723. Its ODBC Data Source Name is oracle723.

```
CREATE SERVER oracle723
CLASS 'oraodbc'
USING 'oracle723';
```

The following example creates a directory access server that only sees files within the directory *c:\temp*:

```
CREATE SERVER diskserver0
CLASS 'directory'
USING 'root=c:\temp';
CREATE EXTERNLOGIN DBA TO diskserver0;
CREATE EXISTING TABLE diskdir0 AT 'diskserver0;;;.';

-- Get a list of those files.
SELECT permissions, file_name, size FROM diskdir0;
```

The following example creates a directory access server that sees nine levels of directories:

```
-- Create a directory server that sees 9 levels of directories.
CREATE SERVER diskserver9
CLASS 'directory'
USING 'ROOT=c:\temp;SUBDIRS=9';
CREATE EXTERNLOGIN DBA TO diskserver9;
CREATE EXISTING TABLE diskdir9 AT 'diskserver9;;;.';
```


CREATE SERVICE statement

Use this statement to permit a database server to act as a web server.

Syntax 1 - DISH service

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2 - SOAP service

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**DATATYPE** { **ON** | **OFF** | **IN** | **OUT** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

Syntax 3 - Miscellaneous services

CREATE SERVICE *service-name*
TYPE { **'RAW'** | **'HTML'** | **'XML'** }
[**URL** [**PATH**] { **ON** | **OFF** | **ELEMENTS** }]
[*common-attributes*]
[**AS** { *statement* | **NULL** }]

common-attributes:
[**AUTHORIZATION** { **ON** | **OFF** }]
[**SECURE** { **ON** | **OFF** }]
[**USER** { *user-name* | **NULL** }]

Parameters

service-name Web service names can be any sequence of alphanumeric characters or /, -, _, ., !, ~, *, ', (, or), except that the first character must not begin with a slash (/) and the name must not contain two or more consecutive slash characters.

Unlike other services, you cannot specify a forward slash (/) in a DISH service name.

AUTHORIZATION clause Determines whether users must specify a user name and password when connecting to the service. If authorization is OFF, the AS clause is required and a single user must be identified by the USER clause. All requests are run using that user's account and permissions.

If authorization is ON, all users must provide a user name and password. Optionally, you can limit the users that are permitted to use the service by providing a user or group name using the USER clause. If the user name is NULL, all users can access the service.

The default value is ON. It is recommended that production systems be run with authorization turned on and that you grant permission to use the service by adding users to a group.

SECURE clause Indicates whether unsecured connections are accepted. ON indicates that only HTTPS connections are to be accepted. Service requests received on the HTTP port are automatically redirected to the HTTPS port. If set to OFF, both HTTP and HTTPS connections are accepted. The default value is OFF.

USER clause If authorization is disabled, this parameter becomes mandatory and specifies the user ID used to execute all service requests. If authorization is enabled (the default), this optional clause identifies the user or group permitted to access the service. The default value is NULL, which grants access to all users.

URL clause Determines whether URL paths are accepted and, if so, how they are processed. OFF indicates that nothing must follow the service name in a URL request. ON indicates that the remainder of the URL is interpreted as the value of a variable named url. ELEMENTS indicates that the remainder of the URL path is to be split at the slash characters into a list of up to 10 elements. The values are assigned to variables named url plus a numeric suffix of between 1 and 10; for example, the first three variable names are url1, url2, and url3. If fewer than 10 values are supplied, the remaining variables are set to NULL. If the service name ends with the character /, then URL must be set to OFF. The default value is OFF.

GROUP clause Applies to DISH services only. Specifies a common prefix that controls which SOAP services the DISH service exposes. For example, specifying GROUP *xyz* exposes only SOAP services *xyz*/*aaaa*, *xyz*/*bbbb*, or *xyz*/*cccc,* but does not expose *abc/aaaa* or *xyzaaaa*. If no group name is specified, the DISH service exposes all the SOAP services in the database. SOAP services can be exposed by more than one DISH service. The same characters are permitted in group names as in service names.

DATATYPE clause Applies to SOAP services only. Controls whether data typing is supported for parameter inputs and/or result set outputs (responses) for all SOAP service formats. When supported, data typing allows a SOAP toolkit to parse and cast the data to the appropriate type. Parameter data types are exposed in the schema section of the Web Service Definition Language (WSDL) generated by the DISH service. Output data types are represented as XML schema type attributes for each column of data.

The following values are permitted for the DATATYPE clause:

- **ON** Support data typing for input parameters and result set responses.
- **OFF** Do not support data typing of input parameters and result set responses (the default).
- **IN** Support data typing of input parameters only.
- **OUT** Support data typing of result set responses only.

For more information on SOAP services, see "Using SOAP services" [*SQL Anywhere Server - Programming*].

FORMAT clause Applies to DISH and SOAP services only. Generates output formats compatible with various types of SOAP clients, such as .NET or Java JAX-RPC. If the format of a SOAP service is not specified, the format is inherited from the service's DISH service declaration. If the DISH service also does not declare a format, it defaults to DNET, which is compatible with .NET clients. A SOAP service that does not declare a format can be used with different types of SOAP clients by defining multiple DISH services, each having a different FORMAT type.

The following formats are supported:

- **DNET** Microsoft DataSet format for use with .NET SOAP clients. DNET is the default FORMAT value and was the only format available before version 9.0.2.
- **CONCRETE** A platform-neutral DataSet format for use with clients such as JAX-RPC, or with clients that automatically generate interfaces based on the format of the returned data structure. Specifying this format type exposes an SimpleDataset element within the WSDL. This element describes the result set

as a containment hierarchy of a rowset composed of an array of rows, each of which contains an array of column elements.

- **XML** A simple XML string format. The DataSet is returned as a string that can be passed to an XML parser. This format is the most portable between SOAP clients.

TYPE clause Type clauses indicate the type of result set returned.

The following result types are supported:

- **RAW** The result set is sent to the client without any further formatting. You can produce formatted documents by generating the required tags explicitly within your procedure.
- **HTML** The result set of a statement or procedure is automatically formatted into an HTML document that contains a table.
- **XML** The result set is returned as XML. If the result set is already XML, no additional formatting is applied. If it is not already XML, it is automatically formatted as XML. The effect is similar to that of using the FOR XML RAW clause in a SELECT statement.
- **SOAP** The result set is returned as a SOAP response. The format of the data is determined by the FORMAT clause. All requests to a SOAP service must be valid SOAP requests, not just a simple HTTP requests. For more information about the SOAP standards, see www.w3.org/TR/SOAP.
- **DISH** A DISH service (Determine SOAP Handler) acts as a proxy for those SOAP services identified by the GROUP clause, and generates a WSDL (Web Services Description Language) file for each of these SOAP services.

  For more information, see "Creating web services" [*SQL Anywhere Server - Programming*].

statement If the statement is NULL, the URL must specify the statement to be executed. Otherwise, the specified SQL statement is the only one that can be executed through the service. The statement is mandatory for SOAP services. The default value is NULL.

It is strongly recommended that all services run in production systems define a statement. The statement can be NULL only if authorization is enabled.

Remarks

The CREATE SERVICE statement causes the database server to act as a web server. A new entry is created in the ISYSWEBSERVICE system table.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER SERVICE statement" on page 323
- "DROP SERVICE statement" on page 506

- "ISYSWEBSERVICE system table" on page 734
- "SQL Anywhere Web Services" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

To set up a web server quickly, start a database server with the -xs option (for example, -xs http), then execute the following statement:

```
CREATE SERVICE tables TYPE 'HTML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTAB;
```

After executing this statement, use any web browser to open the URL http://localhost/tables.

The following example demonstrates how to write a Hello World program.

```
CREATE PROCEDURE hello_world_proc( )
RESULT (html_doc long varchar)
BEGIN
   CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
   SELECT '<html>\n'
       || '<head><title>Hello World</title></head>\n'
       || '<body>\n'
       || '<h1>Hello World!</h1>\n'
       || '</body>\n'
       || '</html>\n';
END;

CREATE SERVICE hello_world TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL hello_world_proc;
```

After executing this statement, use any web browser to open the URL *http://localhost/hello_world*.

CREATE STATISTICS statement

Recreates the column statistics used by the optimizer, and stores them in the ISYSCOLSTAT system table.

Syntax

CREATE STATISTICS *table-name* [**(** *column-list* **)**]

Remarks

The CREATE STATISTICS statement recreates the column statistics that SQL Anywhere uses to optimize database queries, and can be performed on base tables, local temporary tables, and global temporary tables. You cannot create statistics on proxy tables. Column statistics reflect the distribution of data in the database for the specified columns. By default, column statistics are automatically created for tables with five or more rows.

In rare circumstances, when your database queries are very variable, and when data distribution is not uniform or the data is changing frequently, you can improve performance by running the CREATE STATISTICS statement against a table or column.

When executing, the CREATE STATISTICS statement updates existing column statistics regardless of the size of the table, unless the table is empty, in which case nothing is done. If column statistics exist for an empty table, they remain unchanged by the CREATE STATISTICS statement. To remove column statistics for an empty table, execute the DROP STATISTICS statement.

The process of running CREATE STATISTICS performs a complete scan of the table.

The CREATE STATISTICS statement should be used only in rare circumstances.

Permissions

Must have DBA authority.

Side effects

Query plans may change.

See also

- "DROP STATISTICS statement" on page 508
- "ALTER DATABASE statement" on page 301
- "LOAD TABLE statement" on page 585
- "Query Optimization and Execution" [*SQL Anywhere Server - SQL Usage*]
- "CREATE INDEX statement" on page 405
- "ISYSCOLSTAT system table" on page 727

Standards and compatibility

- **SQL/2003** Vendor extension.

CREATE SUBSCRIPTION statement [SQL Remote]

Use this statement to create a subscription for a user to a publication.

Syntax

CREATE SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*

publication-name: *identifier*

subscription-value, *subscriber-id*: *string*

subscriber-id: *string*

Parameters

publication-name The name of the publication to which the user is being subscribed. This can include the owner of the publication.

subscription-value A string that is compared to the subscription expression of the publication. The subscriber receives all rows for which the subscription expression matches the subscription value.

subscriber-id The user ID of the subscriber to the publication. This user must have been granted REMOTE permissions.

Remarks

In a SQL Remote installation, data is organized into publications for replication. To receive SQL Remote messages, a subscription must be created for a user ID with REMOTE permissions.

If a string is supplied in the subscription, it is matched against each SUBSCRIBE BY expression in the publication. The subscriber receives all rows for which the value of the expression is equal to the supplied string.

In SQL Remote, publications and subscriptions are two-way relationships. If you create a subscription for a remote user to a publication on a consolidated database, you should also create a subscription for the consolidated database on the remote database. The extraction utility carries this out automatically.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “DROP SUBSCRIPTION statement [SQL Remote]” on page 509
- “GRANT REMOTE statement [SQL Remote]” on page 556
- “SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]” on page 689
- “START SUBSCRIPTION statement [SQL Remote]” on page 679
- “ISYSSUBSCRIPTION system table” on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a subscription for the user p_chin to the publication pub_sales. The subscriber receives all rows for which the subscription expression has a value of Eastern.

```
CREATE SUBSCRIPTION
TO pub_sales ( 'Eastern' )
FOR p_chin;
```

CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement in a SQL Anywhere remote database to create a subscription between a MobiLink user and a publication.

Syntax

CREATE SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, …]
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, …]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** | **tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

TO clause Specify the name of a publication.

FOR clause Specify one or more MobiLink user names. If you specify more than one user name, a separate subscription is created for each user.

ml_username is a user who is authorized to synchronize with the MobiLink server.

For more information about synchronization user names, see "Introduction to MobiLink users" [*MobiLink - Client Administration*].

Omit the FOR clause to set the protocol type, protocol options, and extended options for a publication.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

TYPE clause This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

For more information about network protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

ADDRESS clause This clause specifies network protocol options such as the location of the MobiLink server. Multiple options must be separated with semi-colons.

For a complete list of protocol options, see "MobiLink client network protocol options" [*MobiLink - Client Administration*].

OPTION clause This clause allows you to set extended options for the subscription. If no FOR clause is provided, the extended options act as default settings for the publication.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

For a complete list of options, see "MobiLink SQL Anywhere Client Extended Options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 328
- "DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 510
- SQL Anywhere MobiLink clients: "Creating synchronization subscriptions" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates a subscription between the MobiLink user ml_user1 and the publication called sales_publication and sets the memory to 3 MB:

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR ml_user1
   OPTION memory='3m';
```

The following example omits the FOR clause, and so stores settings for the publication called sales_publication:

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
```



```
ADDRESS 'host=test.internal;port=2439;
   security=ecc_tls'
OPTION memory='2m';
```


CREATE SYNCHRONIZATION USER statement [MobiLink]

Use this statement in a SQL Anywhere remote database to create a MobiLink user.

Syntax

CREATE SYNCHRONIZATION USER *ml_username*
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, …]

ml_username: *identifier*

network-protocol : **tcpip** | **http** | **https** | **tls**

protocol-options : *string*

value: *string* | *integer*

Parameters

ml_username A name identifying a MobiLink user.

For more information about MobiLink users, see "Introduction to MobiLink users" [*MobiLink - Client Administration*].

TYPE clause This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

ADDRESS clause This clause specifies *protocol-options* in the form *keyword*=*value*, separated by semicolons. Which settings you supply depends on the communication protocol you are using (TCPIP, TLS, HTTP, or HTTPS).

For a complete list of protocol options, see "MobiLink client network protocol options" [*MobiLink - Client Administration*].

OPTION clause The OPTION clause allows you to set extended options using *option=value* in a comma-separated list.

The values for each option cannot contain equal signs or semicolons. The database server accepts any option that you enter without checking for its validity. Therefore, if you misspell an option or enter an invalid value, no error message appears until you run the dbmlsync command to perform synchronization.

Options set for a synchronization user can be overridden in individual subscriptions or on the dbmlsync command line.

For information about extended options, see "MobiLink SQL Anywhere Client Extended Options" [*MobiLink - Client Administration*].

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION USER statement [MobiLink]" on page 330
- "DROP SYNCHRONIZATION USER statement [MobiLink]" on page 511
- "Encrypting MobiLink client/server communications" [*SQL Anywhere Server - Database Administration*]
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates a MobiLink user named SSinger, who synchronizes over TCP/IP with a server computer named mlserver.mycompany.com using the password Sam. The use of a password in the user definition is *not* secure.

```
CREATE SYNCHRONIZATION USER SSinger
TYPE http
ADDRESS 'host=mlserver.mycompany.com'
OPTION MobiLinkPwd='Sam';
```


CREATE TABLE statement

Use this statement to create a new table in the database and, optionally, to create a table on a remote server.

Syntax

CREATE [**GLOBAL TEMPORARY**] **TABLE** [*owner*.]*table-name*
({ *column-definition* | *table-constraint* | *pctfree* }, … **)**
[{ **IN** | **ON** } *dbspace-name*]
[**ENCRYPTED**]
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**
  | **NOT TRANSACTIONAL**]
[**AT** *location-string*]
[**SHARE BY ALL**]

column-definition :
column-name data-type
[**COMPRESSED**]
[**INLINE** { *inline-length* | **USE DEFAULT** }]
[**PREFIX** { *prefix-length* | **USE DEFAULT** }]
[[**NO**] **INDEX**]
[[**NOT**] **NULL**]
[**DEFAULT** *default-value*]
[*column-constraint* …]

default-value :
 special-value
| *string*
| *global variable*
| [**-**] *number*
| **(** *constant-expression* **)**
| *built-in-function***(** *constant-expression* **)**
| **AUTOINCREMENT**
| **CURRENT DATABASE**
| **CURRENT REMOTE USER**
| **CURRENT UTC TIMESTAMP**
| **GLOBAL AUTOINCREMENT** [**(** *partition-size* **)**]
| **NULL**
| **TIMESTAMP**
| **UTC TIMESTAMP**
| **LAST USER**

special-value:
CURRENT {
  DATE
  | **TIME**
  | **TIMESTAMP**
  | **UTC TIMESTAMP**
  | **USER**
  | **PUBLISHER**
}
| **USER**

column-constraint :
[**CONSTRAINT** *constraint-name*] {
  UNIQUE[**CLUSTERED**]


```
  | PRIMARY KEY [ CLUSTERED ] [ ASC | DESC ]
  | REFERENCES table-name [ ( column-name ) ]
      [ MATCH [ UNIQUE ] { SIMPLE | FULL } ]
      [ actions ] [ CLUSTERED ]
    }
| [ CONSTRAINT constraint-name ] CHECK ( condition )
| COMPUTE ( expression )

table-constraint :
[ CONSTRAINT constraint-name ] {
    UNIQUE [ CLUSTERED ] ( column-name [ ASC | DESC ], … )
  | PRIMARY KEY [ CLUSTERED ] ( column-name [ ASC | DESC ], … )
  | CHECK ( condition )
  | foreign-key-constraint
}

foreign-key-constraint :
[ NOT NULL ] FOREIGN KEY [ role-name ]
   [ ( column-name [ ASC | DESC ], … ) ]
   REFERENCES table-name
   [ ( column-name, … ) ]
   [ MATCH [ UNIQUE] { SIMPLE | FULL } ]
   [ actions ] [ CHECK ON COMMIT ] [ CLUSTERED ] [ FOR OLAP WORKLOAD ]

action :
ON { UPDATE | DELETE }
…{ CASCADE | SET NULL | SET DEFAULT | RESTRICT }

location-string :
 remote-server-name.[db-name].[owner].object-name
| remote-server-name;[db-name];[owner];object-name

pctfree : PCTFREE percent-free-space

percent-free-space : integer
```

Parameters

IN clause The IN clause specifies the dbspace in which the table is to be created. If the table is a GLOBAL TEMPORARY table, the IN clause is ignored.

For more information about dbspaces, see "CREATE DBSPACE statement" on page 382.

You can also specify the dbspace in which the table is created by setting the default_dbspace option before executing the CREATE TABLE statement. See "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*].

ENCRYPTED The encrypted clause specifies to encrypt the table. You must enable table encryption when you create a database if you want to encrypt tables. The table is encrypted using the encryption key and algorithm specified at database creation time. See "Enabling table encryption" [*SQL Anywhere Server - Database Administration*].

ON COMMIT clause The ON COMMIT clause is allowed only for temporary tables. By default, the rows of a temporary table are deleted on COMMIT. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified.

NOT TRANSACTIONAL The NOT TRANSACTIONAL clause is allowed when creating a global temporary table. A table created using NOT TRANSACTIONAL is not affected by either COMMIT or ROLLBACK. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified. For information on the benefits of the NOT TRANSACTIONAL clause, see "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*].

AT clause Create a remote table on a different server specified by *location-string*, as well as a proxy table on the current database that maps to the remote table. The AT clause supports the semicolon (;) as a field delimiter in *location-string*. If no semicolon is present, a period is the field delimiter. This allows file names and extensions to be used in the database and owner fields.

For example, the following statement maps the table a1 to the Microsoft Access file *mydbfile.mdb*:

```
CREATE TABLE a1
AT 'access;d:\mydbfile.mdb;;a1';
```

For information on remote servers, see "CREATE SERVER statement" on page 435. For information on proxy tables, see "CREATE EXISTING TABLE statement" on page 395 and "Specifying proxy table locations" [*SQL Anywhere Server - SQL Usage*].

Windows CE does not support the AT clause.

Foreign key definitions are ignored on remote tables. Foreign key definitions on local tables that refer to remote tables are also ignored. Primary key definitions are sent to the remote server if the database server supports primary keys.

SHARE BY ALL clause Use this clause only when creating global temporary tables to allow the table to be shared by all connections to the database. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified.

For information on the characteristics of temporary tables, see "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*].

column-definition Define a column in the table. The following are part of column definitions.

- **column-name** The column name is an identifier. Two columns in the same table cannot have the same name. See "Identifiers" on page 7.
- **data-type** The type of data stored in the column. See "SQL Data Types" on page 47.
- **COMPRESSED** Compress the column. For example, the following statement creates a table, t, with two columns: filename and contents. The contents column is LONG BINARY and is compressed:

  ```
  CREATE TABLE t (
    filename VARCHAR(255),
    contents LONG BINARY COMPRESSED
  );
  ```

- **INLINE and PREFIX**
  The INLINE and PREFIX clauses are for use with storing BLOBs (character or binary data types only). Use the INLINE clause to specify the maximum BLOB size, in bytes, to store in the column. BLOBs that exceed the INLINE value are stored outside of the row in table extension pages. Use the PREFIX

clause to specify how many bytes of the BLOB to duplicate and store with the row. Prefix data can improve performance when processing requests that need only the prefix bytes of a BLOB.

If neither INLINE nor PREFIX is specified, or if USE DEFAULT is specified, default values are applied as follows:

- For character data type columns, such as CHAR, NCHAR, LONG VARCHAR, the default value of INLINE is 256, and the default value of PREFIX is 8.
- For binary data type columns, such as BINARY, LONG BINARY, VARBINARY, BIT, VARBIT, LONG VARBIT, BIT VARYING, and UUID, the default value of INLINE is 256, and the default value of PREFIX is 0.

> **Note**
> It is strongly recommended that you use the default values unless there are specific circumstances that require a different setting. The default values have been chosen to balance performance and disk space requirements. For example, if you set INLINE to a large value, and all the BLOBs are stored inline, row processing performance may degrade. If you set PREFIX too high, you increase the amount of disk space required to store BLOBs since the prefix data is a duplicate of a portion of the BLOB.

If only one of the values is specified, the other value is automatically set to the largest amount that does not conflict with the specified value. Neither the INLINE nor PREFIX value can exceed the database page size. Also, there is a small amount of overhead reserved in a table page that cannot be used to store row data. Therefore, specifying an INLINE value approximate to the database page size can result in a slightly smaller number of bytes being stored inline.

In the case of compressed columns, regardless of the settings of INLINE and PREFIX, the behavior is as though INLINE and PREFIX were set to 0. That is, no prefix is stored, and the BLOB is stored in table extension pages, and if INDEX was specified (the default), BLOB indexing is still performed. If, at a later time, the column is uncompressed, the settings previously in effect for INLINE and PREFIX are restored.

- **[NO] INDEX**
  When storing BLOBs (character or binary types only), use this clause to specify whether to create BLOB indexes. If this clause is not specified, the database server creates the indexes. BLOB indexes can improve performance when random access searches within the BLOBs are required. However, for some types of BLOB values, such as images and multimedia files, BLOB indexing is not required and, in fact, performance can improve if BLOB indexing is turned off. Specify NO INDEX to turn off BLOB indexing for the specified column.

  > **Note**
  > A BLOB index is not the same as a database index. A BLOB index is created to provide faster random access into BLOB data, whereas a database index is created to index values in one or more columns.

- **NOT NULL** If NOT NULL is specified, or if the column is in a UNIQUE or PRIMARY KEY constraint, the column cannot contain NULL in any row.
- **DEFAULT** For more information on the *special-value*, see "Special values" on page 30.

If a DEFAULT value is specified, it is used as the value for the column in any INSERT statement that does not specify a value for the column. If no DEFAULT value is specified, it is equivalent to DEFAULT NULL.
Following is a list of possible values for DEFAULT:

- **Constant expressions** Constant expressions that do not reference database objects are allowed in a DEFAULT clause, so functions such as GETDATE or DATEADD can be used. If the expression is not a function or simple value, it must be enclosed in parentheses.

- **AUTOINCREMENT** When using AUTOINCREMENT, the column must be one of the integer data types, or an exact numeric type.

  On inserts into the table, if a value is not specified for the AUTOINCREMENT column, a unique value larger than any other value in the column is generated. If an INSERT specifies a value for the column that is larger than the current maximum value for the column, that value is inserted and then used as a starting point for subsequent inserts.

  Deleting rows does not decrement the AUTOINCREMENT counter. Gaps created by deleting rows can only be filled by explicit assignment when using an insert. After an explicit insert of a column value less than the maximum, subsequent rows without explicit assignment are still automatically incremented with a value of one greater than the previous maximum.

  You can find the most recently inserted value of the column by inspecting the @@identity global variable.

  AUTOINCREMENT values are maintained as signed 64-bit integers, corresponding to the data type of the max_identity column in the SYSTABCOL system view. When the next value to be generated exceeds the maximum value that can be stored in the column to which the AUTOINCREMENT is assigned, NULL is returned. If the column has been declared to not allow NULLs, as is the case for primary key columns, a SQL error is generated.

  The identity column is a Transact-SQL-compatible alternative to using the AUTOINCREMENT default. In SQL Anywhere, the identity column is implemented as AUTOINCREMENT default. For information, see "The special IDENTITY column" [*SQL Anywhere Server - SQL Usage*].

- **GLOBAL AUTOINCREMENT** This default is intended for use when multiple databases are used in a SQL Remote replication or MobiLink synchronization environment.
  This option is similar to AUTOINCREMENT, except that the domain is partitioned. Each partition contains the same number of values. You assign each copy of the database a unique global database identification number. SQL Anywhere supplies default values in a database only from the partition uniquely identified by that database's number.The partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size can be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.If the column is of type BIGINT or UNSIGNED BIGINT, the default partition size is 2^{32}, = 4294967296; for columns of all other types, the default partition size is 2^{16} = 65536. Since these defaults may be inappropriate, especially if your column is not of type INT or BIGINT, it is best to specify the partition size explicitly.When using this default, the value of the public option global_database_id in each database must be set to a unique, non-negative

integer. This value uniquely identifies the database and indicates from which partition default values are to be assigned. The range of allowed values is *np* + 1 to p(*n* + 1), where *n* is the value of the public option global_database_id and *p* is the partition size. For example, if you define the partition size to be 1000 and set global_database_id to 3, then the range is from 3001 to 4000.If the previous value is less than *p*(*n* + 1), the next default value is one greater than the previous largest value in the column. If the column contains no values, the first default value is *np* + 1. Default column values are not affected by values in the column outside of the current partition; that is, by numbers less than *np* + 1 or greater than *p*(*n* + 1). Such values may be present if they have been replicated from another database via MobiLink or SQL Remote.You can find the most recently inserted value of the column by inspecting the @@identity global variable.GLOBAL AUTOINCREMENT values are maintained as signed 64-bit integers, corresponding to the data type of the max_identity column in the SYSTABCOL system view. When the supply of values within the partition has been exhausted, NULL is returned. If the column has been declared to not allow NULLs, as is the case for primary key columns, a SQL error is generated. In this case, a new value of global_database_id should be assigned to the database to allow default values to be chosen from another partition. To detect that the supply of unused values is low and handle this condition, create an event of type GlobalAutoincrement. See "Understanding events" [*SQL Anywhere Server - Database Administration*].Because the public option global_database_id cannot be set to a negative value, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.If the public option global_database_id is set to the default value of 2147483647, a NULL value is inserted into the column. If NULL values are not permitted, attempting to insert the row causes an error.

- **TIMESTAMP** Provides a way of indicating when each row in the table was last modified. When a column is declared with DEFAULT TIMESTAMP, a default value is provided for inserts, and the value is updated with the current date and time whenever the row is updated.

  To provide a default value on insert, but not update the column whenever the row is updated, use DEFAULT CURRENT TIMESTAMP instead of DEFAULT TIMESTAMP.

  For more information on timestamp columns, see "The special Transact-SQL timestamp column and data type" [*SQL Anywhere Server - SQL Usage*].

  Columns declared with DEFAULT TIMESTAMP contain unique values, so that applications can detect near-simultaneous updates to the same row. If the current timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option. See "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*].

  You can automatically truncate timestamp values in SQL Anywhere based on the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values. See "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*].

  The global variable @@dbts returns a TIMESTAMP value representing the last value generated for a column using DEFAULT TIMESTAMP. See "Global variables" on page 38.

- **string** See "Strings" on page 8.

- **global-variable** See "Global variables" on page 38.

- **column-constraint and table-constraint clauses** Column and table constraints help ensure the integrity of data in the database. If a statement would cause a violation of a constraint, execution of the statement does not complete, any changes made by the statement before error detection are undone, and an error is reported. There are two classes of constraints that can be created: **check constraints**, and **referential integrity (RI) constraints**. Check constraints are used to specify conditions that must be satisfied by values of columns being put into the database. RI constraints establish a relationship between data in different tables that must be maintained in addition to specifying uniqueness requirements for data.

  There are three types of RI constraints: primary key, foreign key, and unique constraint. When you create an RI constraint (primary key, foreign key or unique constraint), the database server enforces the constraint by implicitly creating an index on the columns that make up the key of the constraint. The index is created on the key for the constraint as specified. A key consists of an ordered list of columns and a sequencing of values (ASC/DESC) for each column.

  Constraints can be specified on columns or tables. Generally speaking, a column constraint is one that refers to one column in a table, while a table constraint can refer to one or more columns in a table.

  - **PRIMARY KEY constraint** A primary key uniquely defines each row in the table. Primary keys comprise one or more columns. A table cannot have more than one primary key. In a *column-constraint* clause, specifying PRIMARY KEY indicates that the column is the primary key for the table. In a *table-constraint*, you use the PRIMARY KEY clause to specify one or more columns that, when combined in the specified order, make up the primary key for the table.

    The ordering of columns in a primary key need not match the respective ordinal numbers of the columns. That is, the columns in a primary key need not have the same physical order in the row. Additionally, you cannot specify duplicate column names.

    When you create a primary key, an index for the key is automatically created. You can specify the sequencing of values in the index by specifying ASC (ascending) or DESC (descending) for each column. You can also specify whether to cluster the index, using the CLUSTERED keyword. For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

    Columns included in primary keys cannot allow NULL. Each row in the table has a unique primary key value.

    It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for primary keys. Approximate numeric data types are subject to rounding errors after arithmetic operations.

  - **Foreign key** A foreign key restricts the values for a set of columns to match the values in a primary key or a unique constraint of another table (the primary table). For example, a foreign key constraint could be used to ensure that a customer number in an invoice table corresponds to a customer number in the Customers table. A foreign key constraint can be implemented using a REFERENCES column

constraint (single column only) or a FOREIGN KEY table constraint, in which case the constraint can specify one or more columns.

If you specify *column-name* in a REFERENCES column constraint, it must be a column in the primary table, must be subject to a unique constraint or primary key constraint, and that constraint must consist of only that one column. If you do not specify *column-name*, the foreign key column references the single primary key column of the primary table.

If a specified foreign key column does not exist in the table, the column is created with the same data type as the corresponding column in the primary table. These automatically-created columns cannot be part of the primary key of the foreign table. Thus, a column used in both a primary key and foreign key of the same table must be explicitly created, before the creation of the key.

Foreign key column names are paired with primary key column names according to position in the two lists in a one-to-one manner. If the primary table column names are not specified in a FOREIGN KEY table constraint, then the primary key columns are used. If foreign key column names are not specified, then the foreign key columns are give the same names as the columns in the primary table.

The foreign key column order does not need to reflect the order of columns in the table.

Duplicate column names are not allowed in the foreign key specification.

When you create a foreign key, an index for the key is automatically created. You can specify the sequencing of values in the index by specifying ASC (ascending) or DESC (descending) for each column. You can also specify whether to cluster the index, using the CLUSTERED keyword. For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

- **NOT NULL option** Disallow NULLs in the foreign key columns. A NULL in a foreign key means that no row in the primary table corresponds to this row in the foreign table.

- **role-name clause** The role name is the name of the foreign key. The main function of the role name is to distinguish two foreign keys to the same table. If no role name is specified, the role name is assigned as follows:

  1. If there is no foreign key with a role name the same as the table name, the table name is assigned as the role name.

  2. If the table name is already taken, the role name is the table name concatenated with a zero-padded three-digit number unique to the table.

- **MATCH clause**
  The MATCH clause allows you to control what is considered a match when using a multi-column foreign key. It also allows you to specify uniqueness for the key, thereby eliminating the need to declare uniqueness separately. Following is a list match types you can specify:

- **UNIQUE option** The referencing table can have only one match for non-NULL key values (keys with at least one non-NULL column value are implicitly unique).
- **SIMPLE option** A match occurs for a row in the referencing table if at least one column in the key is NULL, or all the column values match the corresponding column values present in a row of the referenced table.
- **FULL option** A match occurs for a row in the referencing table if all column values in the key are NULL, or if all of the column values match the values present in a row of the referenced table.
- **SIMPLE UNIQUE option** A match occurs if the criteria for both SIMPLE and UNIQUE are met.
- **FULL UNIQUE option** A match occurs if the criteria for both FULL and UNIQUE are met.

- **UNIQUE constraint** In a *column-constraint* clause, a UNIQUE constraint specifies that the values in the column must be unique. In a *table-constraint* clause, the UNIQUE constraint identifies one or more columns that uniquely identify each row in the table. No two rows in the table can have the same values in all the named column(s). A table can have more than one UNIQUE constraint.

  A UNIQUE constraint is not the same as a unique index. Columns of a unique index are allowed to be NULL, while columns in a UNIQUE constraint are not. Also, a foreign key can reference either a primary key or a UNIQUE constraint, but cannot reference a unique index since a unique index can include multiple instances of NULL.

  Columns in a UNIQUE constraint can be specified in any order. Additionally, you can specify the sequencing of values in the corresponding index that is automatically created, by specifying ASC (ascending) or DESC (descending) for each column. You cannot specify duplicate column names, however.

  It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for columns with unique constraints. Approximate numeric data types are subject to rounding errors after arithmetic operations.

  You can also specify whether to cluster the constraint, using the CLUSTERED keyword. For more information about the CLUSTERED option, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

  For information about unique indexes, see "CREATE INDEX statement" on page 405.

- **CHECK constraint** This allows arbitrary conditions to be verified. For example, a CHECK constraint could be used to ensure that a column called Sex only contains the values M or F.

  No row in a table is allowed to violate a CHECK constraint. If an INSERT or UPDATE statement would cause a row to violate the constraint, the operation is not permitted and the effects of the statement are undone. The change is rejected only if a CHECK constraint condition evaluates to FALSE, and the change is allowed if a CHECK constraint condition evaluates to TRUE or UNKNOWN.

For more information about TRUE, FALSE, and UNKNOWN conditions, see "NULL value" on page 43, and "Search conditions" on page 20.

- **COMPUTE clause** The COMPUTE clause is only for use in a *column-constraint* clause. When a column is created using a COMPUTE clause, its value in any row is the value of the supplied expression. Columns created with this constraint are read-only columns for applications: the value is changed by the database server when the expression is evaluated. The COMPUTE expression cannot return a non-deterministic value. For example, it cannot include a special value such as CURRENT TIMESTAMP, or a non-deterministic function.

  The COMPUTE clause is ignored for remote tables.

  Any UPDATE statement that attempts to change the value of a computed column fires any triggers associated with the column.

- **CHECK ON COMMIT option** The CHECK ON COMMIT option overrides the wait_for_commit database option, and causes the database server to wait for a COMMIT before checking RESTRICT actions on a foreign key. The CHECK ON COMMIT option does not delay CASCADE, SET NULL, or SET DEFAULT actions.

  If you use CHECK ON COMMIT without specifying any actions, then RESTRICT is implied as an action for UPDATE and DELETE.

- **FOR OLAP WORKLOAD option** When you specify FOR OLAP WORKLOAD in the REFERENCES clause of a foreign key definition, the database server performs certain optimizations and gather statistics on the key to help improve performance for OLAP workloads, particularly when the optimization_workload is set to OLAP. See "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

  For more information about OLAP, see "OLAP Support" [*SQL Anywhere Server - SQL Usage*].

- **PCTFREE clause** Specifies the percentage of free space you want to reserve for each table page. The free space is used if rows increase in size when the data is updated. If there is no free space in a table page, every increase in the size of a row on that page requires the row to be split across multiple table pages, causing row fragmentation and possible performance degradation.

  The value *percent-free-space* is an integer between 0 and 100. The former specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself. If PCTFREE is not set, or is later dropped, the default PCTFREE value is applied according to the database page size (200 bytes for a 4 KB (and up) page size). The value for PCTFREE is stored in the ISYSTAB system table.

Remarks

The CREATE TABLE statement creates a new table. A table can be created for another user by specifying an owner name. If GLOBAL TEMPORARY is specified, the table is a temporary table. Otherwise, the table is a base table.

Tables created by preceding the table name in a CREATE TABLE statement with a pound sign (#) are declared temporary tables, which are available only in the current connection. See "DECLARE LOCAL TEMPORARY TABLE statement" on page 483.

Columns in SQL Anywhere allow NULLs by default. This setting can be controlled using the allow_nulls_by_default database option. See "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Permissions

Must have RESOURCE authority.

Must have DBA authority to create a table for another user.

Side effects

Automatic commit.

See also

- "CREATE LOCAL TEMPORARY TABLE statement" on page 409
- "ALTER TABLE statement" on page 332
- "CREATE DBSPACE statement" on page 382
- "CREATE EXISTING TABLE statement" on page 395
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 483
- "DROP statement" on page 498
- "Special values" on page 30
- "SQL Data Types" on page 47
- "Creating tables" [*SQL Anywhere Server - SQL Usage*]
- "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature.

  The following are vendor extensions:

  - The { IN | ON } *dbspace-name* clause.
  - The ON COMMIT clause.
  - Some of the default values.

Examples

The following example creates a table for a library database to hold book information.

```
CREATE TABLE library_books (
   -- NOT NULL is assumed for primary key columns
   isbn CHAR(20) PRIMARY KEY,
   copyright_date DATE,
   title CHAR(100),
   author CHAR(50),
   -- column(s) corresponding to primary key of room
   -- are created automatically
```



```
   FOREIGN KEY location REFERENCES room
);
```

The following example creates a table for a library database to hold information on borrowed books. The default value for date_borrowed indicates that the book is borrowed on the day the entry is made. The date_returned column is NULL until the book is returned.

```
CREATE TABLE borrowed_book (
   date_borrowed DATE NOT NULL DEFAULT CURRENT DATE,
   date_returned DATE,
   book CHAR(20)
   REFERENCES library_books (isbn),
   -- The check condition is UNKNOWN until
   -- the book is returned, which is allowed
CHECK( date_returned >= date_borrowed )
);
```

The following example creates tables for a sales database to hold order and order item information.

```
CREATE TABLE Orders (
   order_num INTEGER NOT NULL PRIMARY KEY,
   date_ordered DATE,
   name CHAR(80)
);
CREATE TABLE Order_item (
   order_num INTEGER NOT NULL,
   item_num SMALLINT NOT NULL,
   PRIMARY KEY ( order_num, item_num ),
   -- When an order is deleted, delete all of its
   -- items.
   FOREIGN KEY ( order_num )
   REFERENCES Orders ( order_num )
   ON DELETE CASCADE
);
```

The following example creates a table named t1 at the remote server SERVER_A and creates a proxy table named t1 that is mapped to the remote table.

```
CREATE TABLE t1
( a INT,
  b CHAR(10) )
AT 'SERVER_A.db1.joe.t1';
```


CREATE TRIGGER statement

Use this statement to create a trigger on a table.

Syntax 1

CREATE TRIGGER *trigger-name trigger-type*
{ *trigger-event-list* | **UPDATE OF** *column-list* }
[**ORDER** *integer*] **ON** *table-name*
[**REFERENCING** [**OLD AS** *old-name*]
[**NEW AS** *new-name*]]
[**REMOTE AS** *remote-name*]]
[**FOR EACH** { **ROW** | **STATEMENT** }]
[**WHEN (** *search-condition* **)**]
BEGIN-statement

Syntax 2

CREATE TRIGGER *trigger-name trigger-type*
{ *trigger-event-list* | **UPDATE OF** *column-list* }
[**ORDER** *integer*] **ON** *table-name*
[**REFERENCING** [**OLD AS** *old-name*]
[**NEW AS** *new-name*]]
[**REMOTE AS** *remote-name*]]
[**FOR EACH** { **ROW** | **STATEMENT** }]
[**WHEN (** *search-condition* **)**]
BEGIN [**IF UPDATE (** *column-name* **) THEN**
[{ **AND** | **OR** } **UPDATE (** *column-name* **)**] …]
BEGIN-statement
[**ELSEIF UPDATE (** *column-name* **) THEN**
[{ **AND** | **OR** } **UPDATE (** *column-name* **)**] …]
BEGIN-statement
END IF]
END

column-list : *column-name* [, *column-name*, ...]

trigger-type : **BEFORE** | **AFTER** | **INSTEAD OF** | **RESOLVE**

trigger-event-list : *trigger-event* [, *trigger-event*, ...]

trigger-event : **DELETE** | **INSERT** | **UPDATE**

Parameters

Trigger-event Triggers can be fired by the following events. You can define either multiple trigger events listed, or one UPDATE OF column-list event:

- **DELETE** Invoked whenever a row of the associated table is deleted.
- **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
- **UPDATE** Invoked whenever a row of the associated table is updated.

- **UPDATE OF column-list** Invoked whenever a row of the associated table is updated and a column in the *column-list* is modified. This type of trigger event cannot be used in a *trigger-event-list*; it must be the only trigger event defined for the trigger. This clause cannot be used in an INSTEAD OF trigger.

You can write separate triggers for each event that you need to handle or, if you have some shared actions and some actions that depend on the event, you can create a trigger for all events and use an IF statement to distinguish the action taking place. See "IF statement" on page 563.

trigger-type Row-level triggers can be defined to execute BEFORE, AFTER, or INSTEAD OF an insert, update, or delete operation. Statement-level triggers can be defined to execute INSTEAD OF or AFTER the statement.

INSTEAD OF triggers are the only form of trigger that you can be define on a (non-materialized) view. If you attempt to define a BEFORE or AFTER trigger on a view, a SQLE_INVALID_TRIGGER_VIEW error is returned

INSTEAD OF triggers replace the triggering action with another action. When an INSTEAD OF trigger fires, the triggering action is skipped and the specified action is performed instead. INSTEAD OF triggers can be defined as row-level or statement-level. A statement-level INSTEAD OF trigger replaces the entire statement, including all row-level operations. If a statement-level INSTEAD OF trigger fires, no row-level triggers fire as a result of that statement. However, the body of the statement-level trigger could perform other operations that, in turn, cause other row-level triggers to fire.

If you are defining an INSTEAD OF trigger, you cannot use the UPDATE OF *column-list* clause, the ORDER clause, or the WHEN clause.

For more information on the capabilities of, and restrictions for, INSTEAD OF triggers, see "INSTEAD OF triggers" [*SQL Anywhere Server - SQL Usage*].

BEFORE UPDATE triggers fire any time an UPDATE occurs on a row, whether or not the new value differs from the old value. AFTER UPDATE triggers fire only if the new value is different from the old value.

The RESOLVE trigger type is for use with SQL Remote: it fires before row-level UPDATE or UPDATE OF *column-list* only.

FOR EACH clause To declare a trigger as a row-level trigger, use the FOR EACH ROW clause. To declare a trigger as a statement-level trigger, you can either use a FOR EACH STATEMENT clause or omit the FOR EACH clause. For clarity, it is recommended that you enter the FOR EACH STATEMENT clause if declaring a statement-level trigger.

ORDER clause Triggers of the same type (insert, update, or delete) that fire at the same time (before, after, or resolve) can use the ORDER clause to determine the order that the triggers are fired. Specifying ORDER 0 is equivalent to omitting the ORDER clause. This clause cannot be used in an INSTEAD OF trigger; there can only be one INSTEAD OF trigger of each type (insert, update, or delete) defined on a table or view.

REFERENCING clause The REFERENCING OLD and REFERENCING NEW clauses allow you to refer to the inserted, deleted or updated rows. For the purposes of this clause, an UPDATE is treated as a delete followed by an insert.

An INSERT takes the REFERENCING NEW clause, which represents the inserted row. There is no REFERENCING OLD clause.

A DELETE takes the REFERENCING OLD clause, which represents the deleted row. There is no REFERENCING NEW clause.

An UPDATE takes the REFERENCING OLD clause, which represents the row before the update, and it takes the REFERENCING NEW clause, which represents the row after the update.

The meaning of REFERENCING OLD and REFERENCING NEW differs, depending on whether the trigger is a row-level or a statement-level trigger. For row-level triggers, the REFERENCING OLD clause allows you to refer to the values in a row prior to an update or delete, and the REFERENCING NEW clause allows you to refer to the inserted or updated values. The OLD and NEW rows can be referenced in BEFORE and AFTER triggers. The REFERENCING NEW clause allows you to modify the new row in a BEFORE trigger before the insert or update operation takes place.

For statement-level triggers, the REFERENCING OLD and REFERENCING NEW clauses refer to declared temporary tables holding the old and new values of the rows. The default names for these tables are deleted and inserted.

The REFERENCING REMOTE clause is for use with SQL Remote. It allows you to refer to the values in the VERIFY clause of an UPDATE statement. It should be used only with RESOLVE UPDATE or RESOLVE UPDATE OF column-list triggers.

WHEN clause The trigger fires only for rows where the search-condition evaluates to true. The WHEN clause can be used only with row level triggers. This clause cannot be used in an INSTEAD OF trigger.

Remarks

The CREATE TRIGGER statement creates a trigger associated with a table in the database, and stores the trigger in the database.

You cannot define a trigger on a materialized view. If you do, an SQLE_INVALID_TRIGGER_MATVIEW error is returned.

The trigger is declared as either a row-level trigger, in which case it executes before or after each row is modified, or as a statement-level trigger, in which case it executes after the entire triggering statement is completed.

Permissions

Must have RESOURCE authority and have ALTER permissions on the table, or must be the owner of the table or have DBA authority. CREATE TRIGGER puts a table lock on the table, and thus requires exclusive use of the table.

Side effects

Automatic commit.

See also

- “BEGIN statement” on page 351
- “CREATE PROCEDURE statement” on page 414
- “CREATE TRIGGER statement [T-SQL]” on page 468
- “DROP statement” on page 498
- “Using Procedures, Triggers, and Batches” [*SQL Anywhere Server - SQL Usage*]
- “UPDATE statement” on page 703

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. Some clauses are vendor extensions.

Example

The first example creates a row-level trigger. When a new department head is appointed, update the ManagerID column for employees in that department.

```
CREATE TRIGGER TR_change_managers
BEFORE UPDATE OF DepartmentHeadID
ON Departments
REFERENCING OLD AS old_dept NEW AS new_dept
FOR EACH ROW
BEGIN
   UPDATE Employees
   SET Employees.ManagerID=new_dept.DepartmentHeadID
   WHERE Employees.DepartmentID=old_dept.DepartmentID
END;
```

The next example, which is more complex, deals with a statement-level trigger. First, create a table as follows:

```
CREATE TABLE t0
( id integer NOT NULL,
 times timestamp NULL DEFAULT current timestamp,
 remarks text NULL,
 PRIMARY KEY ( id )
);
```

Next, create a statement-level trigger for this table:

```
CREATE TRIGGER insert-st AFTER INSERT ORDER 4 ON
t0
REFERENCING NEW AS new_name
FOR EACH STATEMENT
BEGIN
  DECLARE @id1 INTEGER;
  DECLARE @times1 TIMESTAMP;
  DECLARE @remarks1 LONG VARCHAR;
  DECLARE @err_notfound EXCEPTION FOR SQLSTATE VALUE '02000';
//declare a cursor for table new_name
  DECLARE new1 CURSOR FOR
   SELECT id, times, remarks FROM
      new_name;
  OPEN new1;
 //Open the cursor, and get the value
  LoopGetRow:
  LOOP
      FETCH NEXT new1
   INTO @id1, @times1,@remarks1;
      IF SQLSTATE = @err_notfound THEN
   LEAVE LoopGetRow
      END IF;
      //print the value or for other use
      PRINT (@remarks1);
  END LOOP LoopGetRow;
  CLOSE new1
END;
```

The next example shows how you can use REFERENCING NEW in a BEFORE UPDATE trigger. This example ensures that postal codes in the new Employees table are in uppercase:


```
CREATE TRIGGER emp_upper_postal_code
BEFORE UPDATE OF PostalCode
ON Employees
REFERENCING NEW AS new_emp
FOR EACH ROW
BEGIN
   -- Ensure postal code is uppercase (employee might be
   -- in Canada where postal codes contain letters)
   SET new_emp.PostalCode = UPPER(new_emp.PostalCode)
END;
```

The next example shows how you can use REFERENCING OLD in a BEFORE DELETE trigger. This example prevents deleting an employee from the Employees table who has not been terminated.

```
CREATE TRIGGER TR_check_delete_employee
BEFORE DELETE
ON Employees
REFERENCING OLD AS current_employees
FOR EACH ROW /* WHEN( search_condition ) */
BEGIN
   IF current_employees.TerminationDate IS NULL THEN
    RAISERROR 30001 'You cannot delete an employee who has not been fired';
    END IF;
END
```

The next example shows how you can use REFERENCING NEW and REFERENCING OLD in a BEFORE UPDATE trigger. This example prevents a decrease in an employee's salary.

```
CREATE TRIGGER TR_check_salary_decrease
    BEFORE UPDATE
      ON Employees
    REFERENCING OLD AS before_update
    NEW AS after_update
FOR EACH ROW
BEGIN
   IF after_update.salary < before_update.salary THEN
    RAISERROR 30002 'You cannot decrease a salary';
    END IF;
END
```

The next example shows how you can use REFERENCING NEW and REFERENCING OLD in a BEFORE UPDATE trigger. This example also disallows decreasing an employee's salary, but this trigger is more efficient because it fires only when the salary column is updated.

```
CREATE TRIGGER TR_check_salary_decrease_column
    BEFORE UPDATE OF Salary
      ON Employees
    REFERENCING OLD AS before_update
    NEW AS after_update
FOR EACH ROW /* WHEN( search_condition ) */
BEGIN
   IF after_update.salary < before_update.salary THEN
    RAISERROR 30002 'You cannot decrease a salary';
End IF;
END;
```

The next example shows how you can use REFERENCING NEW and in a BEFORE INSERT and UPDATE trigger. The following example creates a trigger that will fire before a row in the SalesOrderItems table is inserted or updated.

```
CREATE TRIGGER TR_update_date
   BEFORE INSERT, UPDATE
```



```
     ON SalesOrderItems
   REFERENCING NEW AS new_row
FOR EACH ROW
BEGIN
   SET new_row.ShipDate = CURRENT TIMESTAMP;
END
```


CREATE TRIGGER statement [T-SQL]

Use this statement to create a new trigger in the database in a manner compatible with Adaptive Server Enterprise.

Syntax 1

CREATE TRIGGER [*owner*.]*trigger_name*
ON [*owner*.]*table_name*
FOR { **INSERT**, **UPDATE**, **DELETE** }
AS *statement-list*

Syntax 2

CREATE TRIGGER [*owner*.]*trigger_name*
ON [*owner*.]*table_name*
FOR {**INSERT**, **UPDATE**}
AS
[**IF UPDATE (** *column_name* **)**
[{ **AND** | **OR** } **UPDATE (** *column_name* **)**] …]
statement-list
[**IF UPDATE (** *column_name* **)**
[{ **AND** | **OR**} **UPDATE (** *column_name* **)**] …]
statement-list

Remarks

The rows deleted or inserted are held in two temporary tables. In the Transact-SQL form of triggers, they can be accessed using the table names deleted, and inserted, as in Adaptive Server Enterprise. In the Watcom-SQL CREATE TRIGGER statement, these rows are accessed using the REFERENCING clause.

Trigger names must be unique in the database.

Transact-SQL triggers are executed AFTER the triggering statement.

Permissions

Must have RESOURCE authority and have ALTER permissions on the table, or must have DBA authority.

CREATE TRIGGER locks all the rows on the table, and thus requires exclusive use of the table.

Side effects

Automatic commit.

See also

- "CREATE TRIGGER statement" on page 462

Standards and compatibility

- **SQL/2003** Vendor extension.

CREATE VARIABLE statement

Use this statement to create a SQL variable.

Syntax

CREATE VARIABLE *identifier data-type*

Remarks

The CREATE VARIABLE statement creates a new variable of the specified data type. The variable contains the NULL value until it is assigned a different value by the SET statement.

A variable can be used in a SQL expression anywhere a column name is allowed. Name resolution is performed as follows:

1. Match any aliases specified in the query's SELECT list.
2. Match column names for any referenced tables.
3. Assume the name is a variable.

Variables belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement. Variables are not visible to other connections. Variables are not affected by COMMIT or ROLLBACK statements.

Variables are useful for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs.

Local variables in procedures and triggers are declared within a compound statement (see "Using compound statements" [*SQL Anywhere Server - SQL Usage*]).

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 351
- "SQL Data Types" on page 47
- "DROP VARIABLE statement" on page 512
- "SET statement" on page 656
- "VAREXISTS function [Miscellaneous]" on page 278

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

This example creates a variable called first_name, of data type VARCHAR(50).


```
CREATE VARIABLE first_name VARCHAR(50);
```

This example creates a variable called 'birthday', of data type DATE.

```
CREATE VARIABLE birthday DATE;
```


CREATE VIEW statement

Use this statement to create a view on the database. Views are used to give a different perspective on the data, even though it is not stored that way.

Syntax

CREATE VIEW
[*owner*.]*view-name* [**(** *column-name*, … **)**]
AS *select-statement*
[**WITH CHECK OPTION**]

Parameters

view-name The *view-name* is an identifier. The default owner is the current user ID.

column-name The columns in the view are given the names specified in the *column-name* list. If the column name list is not specified, the view columns are given names from the select list items. To use the names from the select list items, each item must be a simple column name or have an alias-name specified (see "SELECT statement" on page 648). All items in the select list must have unique names.

AS clause The SELECT statement on which the view is based. The SELECT statement must not refer to local temporary tables. Also, the SELECT statement can have an ORDER BY or GROUP BY clause, and can be a UNION. However, note that in some cases, particularly when combined with the FIRST or TOP clause, using a SELECT with an ORDER BY clause does affect the results of a view definition.

WITH CHECK OPTION clause The WITH CHECK OPTION clause rejects any updates and inserts to the view that do not meet the criteria of the views as defined by its SELECT statement.

Remarks

The CREATE VIEW statement creates a view with the given name. You can create a view owned by another user by specifying the owner. You must have DBA authority to create a view for another user.

A view name can be used in place of a table name in SELECT, DELETE, UPDATE, and INSERT statements. Views, however, do not physically exist in the database as tables. They are derived each time they are used. The view is derived as the result of the SELECT statement specified in the CREATE VIEW statement. Table names used in a view should be qualified by the user ID of the table owner. Otherwise, a different user ID might not be able to find the table or might get the wrong table.

Views can be updated unless the SELECT statement defining the view contains a GROUP BY clause, an aggregate function, or involves a UNION statement. An update to the view causes the underlying table(s) to be updated.

Typically, a view references tables and views (and their respective attributes) that are defined in the catalog. However, a view can also reference SQL variables. In this case, when a query that references the view is executed, the value of the SQL variable is used. Views that reference SQL variables are called **parameterized views** since the variables act as parameters to the execution of the view.

Parameterized views offer an alternative to embedding the body of an equivalent SELECT block in a query as a derived table in the query's FROM clause. Parameterized views can be especially useful for queries embedded in stored procedures where the SQL variables referenced in the view are input parameters to the procedure.

It is not necessary for the SQL variable to exist when the CREATE VIEW statement is executed. However, if the SQL variable is not defined when a query that refers to the view is executed, a Column Not Found error is returned.

Permissions

Must have RESOURCE authority and SELECT permission on the tables in the view definition.

Side effects

Automatic commit.

See also

- "DROP statement" on page 498
- "CREATE TABLE statement" on page 450
- "CREATE MATERIALIZED VIEW statement" on page 411

Standards and compatibility

- **SQL/2003** Core feature. However, parameterized views are a vendor extension.

Example

The following example creates a view showing information for male employees only. This view has the same column names as the base table.

```
CREATE VIEW MaleEmployees
AS SELECT *
FROM Employees
WHERE Sex = 'M';
```

The following example creates a view showing employees and the departments they belong to.

```
CREATE VIEW EmployeesAndDepartments
  AS SELECT Surname, GivenName, DepartmentName
  FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID;
```

The following example creates a parameterized view based on the variables var1 and var2, which are neither attributes of the Employees nor Departments tables:

```
CREATE VIEW EmployeesByState
  AS SELECT Surname, GivenName, DepartmentName
  FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Employees.State = var1 and Employees.Status = var2;
```

Variables can appear in the view's SELECT statement in any context where a variable is a permitted expression. For example, the following parameterized view utilizes the parameter var1 as the pattern for a LIKE predicate:

```
CREATE VIEW ProductsByDescription
  AS SELECT *
  FROM Products
  WHERE Products.Description LIKE var1;
```

To use this view, the variable var1 must be defined before the query referencing the view is executed. For example, the following could be placed in a procedure, function, or a batch statement:


```
BEGIN
DECLARE var1 CHAR(20);
SET var1 = '%cap%';
SELECT * FROM ProductsByDescription
END
```


DEALLOCATE statement

This statement has no effect in SQL Anywhere, and is ignored. It is provided for compatibility with Adaptive Server Enterprise and Microsoft SQL Server. Refer to your Adaptive Server Enterprise or Microsoft SQL Server documentation for more information about this statement.

Standards and compatibility

- **SQL/2003** Vendor extension.

DEALLOCATE DESCRIPTOR statement [ESQL]

Use this statement to free memory associated with a SQL descriptor area.

Syntax

DEALLOCATE DESCRIPTOR *descriptor-name*

descriptor-name : *string*

Remarks

Frees all memory associated with a descriptor area, including the data items, indicator variables, and the structure itself.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 299
- "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*]
- "SET DESCRIPTOR statement [ESQL]" on page 662

Standards and compatibility

- **SQL/2003** Core feature.

Example

For an example, see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 299.

Declaration section [ESQL]

Use this statement to declare host variables in an embedded SQL program. Host variables are used to exchange data with the database.

Syntax

EXEC SQL BEGIN DECLARE SECTION;
C declarations
EXEC SQL END DECLARE SECTION;

Remarks

A declaration section is simply a section of C variable declarations surrounded by the BEGIN DECLARE SECTION and END DECLARE SECTION statements. A declaration section makes the SQL preprocessor aware of C variables that are used as host variables. Not all C declarations are valid inside a declaration section. See "Using host variables" [*SQL Anywhere Server - Programming*] for more information.

Permissions

None.

See also

- "BEGIN statement" on page 351

Standards and compatibility

- **SQL/2003** Core feature.

Example

```
EXEC SQL BEGIN DECLARE SECTION;
char *surname, initials[5];
int dept;
EXEC SQL END DECLARE SECTION;
```


DECLARE statement

Use this statement to declare a SQL variable within a compound statement (BEGIN … END).

Syntax

DECLARE *variable-name data-type*

Remarks

Variables used in the body of a procedure, trigger, or batch can be declared using the DECLARE statement. The variable persists for the duration of the compound statement in which it is declared.

The body of a Watcom-SQL procedure or trigger is a compound statement, and variables must be declared with other declarations, such as a cursor declaration (DECLARE CURSOR), immediately following the BEGIN keyword. In a Transact-SQL procedure or trigger, there is no such restriction.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "DECLARE CURSOR statement [T-SQL]" on page 482

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following batch illustrates the use of the DECLARE statement and prints a message on the Server Messages window:

```
BEGIN
  DECLARE varname CHAR(61);
  SET varname = 'Test name';
  MESSAGE varname;
END
```

DECLARE CURSOR statement [ESQL] [SP]

Use this statement to declare a cursor. Cursors are the primary means for manipulating the results of queries.

Syntax 1 [ESQL]

DECLARE *cursor-name*
[**UNIQUE**]
[**NO SCROLL**
| **DYNAMIC SCROLL**
| **SCROLL**
| **INSENSITIVE**
| **SENSITIVE**
]
CURSOR FOR
{ *select-statement*
| *statement-name* [**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
| *call-statement* }

Syntax 2 [SP]

DECLARE *cursor-name*
[**NO SCROLL**
| **DYNAMIC SCROLL**
| **SCROLL**
| **INSENSITIVE**
| **SENSITIVE**
]
CURSOR
{ **FOR** *select-statement* [**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
| **FOR** *call-statement*
| **USING** *variable-name* }

cursor-name : *identifier*

statement-name : *identifier* | *hostvar*

variable-name : *identifier*

cursor-concurrency :
BY { **VALUES** | **TIMESTAMP** | **LOCK** }

Parameters

UNIQUE When a cursor is declared UNIQUE, the query is forced to return all the columns required to uniquely identify each row. Often this means ensuring that all columns in the primary key or a uniqueness table constraint are returned. Any columns that are required but were not specified in the query are added to the result set.

A DESCRIBE done on a UNIQUE cursor sets the following additional options in the indicator variables:

- **DT_KEY_COLUMN** The column is part of the key for the row
- **DT_HIDDEN_COLUMN** The column was added to the query because it was required to uniquely identify the rows

NO SCROLL A cursor declared NO SCROLL is restricted to moving forward through the result set using FETCH NEXT and FETCH RELATIVE 0 seek operations.

As rows cannot be returned to once the cursor leaves the row, there are no sensitivity restrictions on the cursor. Consequently, when a NO SCROLL cursor is requested, SQL Anywhere supplies the most efficient kind of cursor, which is an asensitive cursor. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

DYNAMIC SCROLL DYNAMIC SCROLL is the default cursor type. DYNAMIC SCROLL cursors can use all formats of the FETCH statement.

When a DYNAMIC SCROLL cursor is requested, SQL Anywhere supplies an asensitive cursor. When using cursors there is always a trade-off between efficiency and consistency. Asensitive cursors provide efficient performance at the expense of consistency. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

SCROLL A cursor declared SCROLL can use all formats of the FETCH statement. When a SCROLL cursor is requested, SQL Anywhere supplies a value-sensitive cursor. See "Value-sensitive cursors" [*SQL Anywhere Server - Programming*].

SQL Anywhere must execute value-sensitive cursors in such a way that result set membership is guaranteed. DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required.

INSENSITIVE A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor. See "Insensitive cursors" [*SQL Anywhere Server - Programming*].

SENSITIVE A cursor declared SENSITIVE is sensitive to changes to membership or values of the result set. See "Sensitive cursors" [*SQL Anywhere Server - Programming*].

FOR statement-name Statements are named using the PREPARE statement. Cursors can be declared only for a prepared SELECT or CALL.

FOR UPDATE | READ ONLY A cursor declared FOR READ ONLY cannot be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.

Cursors default to FOR UPDATE for single-table queries without an ORDER BY clause, or if the ansi_update_constraints option is set to Off. When the ansi_update_constraints option is set to Cursors or Strict, then cursors over a query containing an ORDER BY clause default to READ ONLY. However, you can explicitly mark cursors as updatable using the FOR UPDATE clause. Because it is expensive to allow updates over cursors with an ORDER BY clause or a join, cursors over a query containing a join of two or more tables are READ ONLY and cannot be made updatable.

In response to any request for a cursor that specifies FOR UPDATE, SQL Anywhere provides either a value-sensitive cursor or a sensitive cursor. Insensitive and asensitive cursors are not updatable.

USING variable-name For use within stored procedures only. The variable is a string containing a SELECT statement for the cursor. The variable must be available when the DECLARE is processed, and so must be one of the following:

- A parameter to the procedure. For example,

```
CREATE FUNCTION GetRowCount( IN qry LONG VARCHAR)
RETURNS INT
BEGIN
  DECLARE crsr CURSOR USING qry;
  DECLARE rowcnt INT;

  SET rowcnt = 0;
  OPEN crsr;
  lp: LOOP
    fetch crsr;
    IF SQLCODE <> 0 THEN LEAVE lp END IF;
    SET rowcnt = rowcnt + 1;
  END LOOP;
  RETURN rowcnt;
END;
```

- Nested inside another BEGIN… END after the variable has been assigned a value. For example,

```
CREATE PROCEDURE get_table_name(
  IN id_value INT, OUT tabname CHAR(128)
)
BEGIN
  DECLARE qry LONG VARCHAR;

  SET qry = 'SELECT table_name FROM SYS.SYSTAB ' ||
            'WHERE table_id=' || string(id_value);
  BEGIN
    DECLARE crsr CURSOR USING qry;
    OPEN crsr;
    FETCH crsr INTO tabname;
    CLOSE crsr;
  END
END;
```

BY VALUES | TIMESTAMP | LOCK In embedded SQL, a concurrency specification can be set by including syntax within the SELECT statement itself, or in the cursor declaration. Pessimistic or optimistic concurrency can be chosen at the cursor level either through options with DECLARE CURSOR or FOR statements, or though the concurrency setting API in specific programming interfaces. If a statement is updatable and the cursor does not specify a particular concurrency control mechanism, the statement's specification is used. The syntax is as follows:

- **FOR UPDATE BY LOCK** The database server acquires intent row locks on FETCHed rows of the result set. These are long-term locks that are held until transaction COMMIT or ROLLBACK.
- **FOR UPDATE BY { VALUES | TIMESTAMP }** The database server utilizes a keyset-driven cursor to enable the application to be informed when rows have been modified or deleted as the result set is scrolled.

Remarks

The DECLARE CURSOR statement declares a cursor with the specified name for a SELECT statement or a CALL statement. In a Watcom-SQL procedure, trigger, or batch, a DECLARE CURSOR statement must

appear with other declarations, immediately following the BEGIN keyword. Also, cursor names must be unique.

If a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

Permissions

None.

Side effects

None.

See also

- "PREPARE statement [ESQL]" on page 610
- "OPEN statement [ESQL] [SP]" on page 601
- "EXPLAIN statement [ESQL]" on page 524
- "SELECT statement" on page 648
- "CALL statement" on page 357
- "FOR statement" on page 530

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following example illustrates how to declare a scroll cursor in embedded SQL:

```
EXEC SQL DECLARE cur_employee SCROLL CURSOR
FOR SELECT * FROM Employees;
```

The following example illustrates how to declare a cursor for a prepared statement in embedded SQL:

```
EXEC SQL PREPARE employee_statement
FROM 'SELECT Surname FROM Employees';
EXEC SQL DECLARE cur_employee CURSOR
FOR employee_statement;
```

The following example illustrates the use of cursors in a stored procedure:

```
BEGIN
  DECLARE cur_employee CURSOR FOR
      SELECT Surname
      FROM Employees;
  DECLARE name CHAR(40);
  OPEN cur_employee;
  lp: LOOP
    FETCH NEXT cur_employee INTO name;
    IF SQLCODE <> 0 THEN LEAVE lp END IF;
    ...
  END LOOP;
  CLOSE cur_employee;
END
```


DECLARE CURSOR statement [T-SQL]

Use this statement to declare a cursor in a manner compatible with Adaptive Server Enterprise.

Syntax

DECLARE *cursor-name*
CURSOR FOR *select-statement*
[**FOR** { **READ ONLY** | **UPDATE** }]

cursor-name : *identifier*

select-statement : *string*

Remarks

DECLARE CURSOR statements in Transact-SQL procedures are treated as executable statements and can appear anywhere in a procedure. The cursor declaration takes effect when the statement is executed and remains in effect until a DEALLOCATE CURSOR statement is executed or until the procedure completes.

In SQL Anywhere, if a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

In a Transact-SQL procedure, trigger, or batch, a DECLARE CURSOR statement can appear after other executable statements.

Permissions

None.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478

Standards and compatibility

- **SQL/2003** Core feature. The FOR UPDATE and FOR READ ONLY options are Transact-SQL extensions.

DECLARE LOCAL TEMPORARY TABLE statement

Use this statement to declare a local temporary table.

Syntax

DECLARE LOCAL TEMPORARY TABLE *table-name*
({ *column-definition* [*column-constraint* …] | *table-constraint* | *pctfree* }, … **)**
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**
| **NOT TRANSACTIONAL**]

pctfree : **PCTFREE** *percent-free-space*

percent-free-space : *integer*

Parameters

For definitions of *column-definition*, *column-constraint*, *table-constraint*, and *pctfree*, see "CREATE TABLE statement" on page 450.

ON COMMIT By default, the rows of a temporary table are deleted on a COMMIT. You can use the ON COMMIT clause to preserve rows on a COMMIT.

NOT TRANSACTIONAL A table created using this clause is not affected by either COMMIT or ROLLBACK. The NOT TRANSACTIONAL clause provides performance improvements in some circumstances because operations on non-transactional temporary tables do not cause entries to be made in the rollback log. For example, NOT TRANSACTIONAL can be useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

Remarks

The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table.

The rows of a declared temporary table are deleted when the table is explicitly dropped or when the table goes out of scope. You can also explicitly delete rows using TRUNCATE or DELETE.

Declared local temporary tables within compound statements exist within the compound statement. (See "Using compound statements" [*SQL Anywhere Server - SQL Usage*]). Otherwise, the declared local temporary table exists until the end of the connection.

If you want a procedure to create a local temporary table that persists after the procedure completes, use the CREATE LOCAL TEMPORARY TABLE statement instead. (See "CREATE LOCAL TEMPORARY TABLE statement" on page 409).

Permissions

None.

Side effects

None.

See also

- "CREATE TABLE statement" on page 450

- "CREATE LOCAL TEMPORARY TABLE statement" on page 409
- "Using compound statements" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Example

The following example illustrates how to declare a temporary table in a stored procedure:

```
BEGIN
  DECLARE LOCAL TEMPORARY TABLE TempTab ( number INT );
  ...
END
```


DELETE statement

Use this statement to delete rows from the database.

Syntax

DELETE [**FIRST** | **TOP** *n*]
[**FROM**] [*owner*.]*table-name*
[**FROM** *table-list*]
[**WHERE** *search-condition*]
[**ORDER BY** { *expression* | *integer* } [**ASC** | **DESC**], …]
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Remarks

Deleting a significant amount of data using the DELETE statement causes an update to column statistics.

The DELETE statement can be used on views, provided the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION statement.

FIRST or TOP clause Primarily for use with the ORDER BY clause, this clause allows you to delete only a certain subset of the rows that satisfy the WHERE clause. The TOP value must be an integer constant or integer variable with value greater than or equal to 0. You cannot use a variable as input with TOP.

FROM clause The FROM clause indicates the table from which to delete rows. The second FROM clause in the DELETE statement qualifies the rows to be deleted from the specified table based on joins. If the second FROM clause is present, the WHERE clause qualifies the rows of this second FROM clause.

The second FROM clause can contain arbitrary complex table expressions, such as KEY and NATURAL joins. For a full description of the FROM clause and joins, see "FROM clause" on page 535.

The following statement illustrates a potential ambiguity in table names in DELETE statements with two FROM clauses that use correlation names:

```
DELETE
FROM table_1
FROM table_1 AS alias_1, table_2 AS alias_2
WHERE ...
```

The table table_1 is identified without a correlation name in the first FROM clause, but with a correlation name in the second FROM clause. In this case, table_1 in the first clause is identified with alias_1 in the second clause—there is only one instance of table_1 in this statement.

This is an exception to the general rule that where a table is identified with a correlation name and without a correlation name in the same statement, two instances of the table are considered.

Consider the following example:

```
DELETE
FROM table_1
FROM table_1 AS alias_1, table_1 AS alias_2
WHERE ...
```

In this case, there are two instances of table_1 in the second FROM clause. The statement will fail with a syntax error as it is ambiguous which instance of the table_1 from the second FROM clause matches the first instance of table_1 in the first FROM clause.

WHERE clause The DELETE statement deletes all the rows that satisfy the conditions in the WHERE clause. If no WHERE clause is specified, all rows from the named table are deleted. If a second FROM clause is present, the WHERE clause qualifies the rows of the second FROM clause.

ORDER BY clause Specifies the sort order for the rows. Normally, the order in which rows are updated does not matter. However, in conjunction with the FIRST or TOP clause the order can be significant.

You cannot use ordinal column numbers in the ORDER BY clause.

Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order.

OPTION clause
This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

  For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].
- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

- "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
- "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
- "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
- "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Permissions

Must have DELETE permission on the table.

Side effects

None.

See also

- "TRUNCATE TABLE statement" on page 693
- "INSERT statement" on page 573
- "INPUT statement [Interactive SQL]" on page 568
- "FROM clause" on page 535

Standards and compatibility

- **SQL/2003** Core feature. The use of more than one table in the FROM clause is a vendor extension.

Example

Remove employee 105 from the database.

```
DELETE
FROM Employees
WHERE EmployeeID = 105;
```

Remove all data prior to 2000 from the FinancialData table.

```
DELETE
FROM FinancialData
WHERE Year < 2000;
```

Remove the first 10 orders from SalesOrderItems table where ship date is older than 2001-01-01 and their region is Central.

```
DELETE TOP 10
FROM SalesOrderItems
FROM SalesOrders
WHERE SalesOrderItems.ID = SalesOrders.ID
  and ShipDate < '2001-01-01' and Region ='Central'
ORDER BY ShipDate ASC;
```

Remove department 600 from the database, executing the statement at isolation level 3.

```
DELETE FROM Departments
WHERE DepartmentID = 600
OPTION( isolation_level = 3 );
```


DELETE (positioned) statement [ESQL] [SP]

Use this statement to delete the data at the current location of a cursor.

Syntax

DELETE [**FROM** *table-spec*] **WHERE CURRENT OF** *cursor-name*

cursor-name : *identifier* | *hostvar*

table-spec : [*owner*.]*correlation-name*

owner : *identifier*

Remarks

This form of the DELETE statement deletes the current row of the specified cursor. The current row is defined to be the last row fetched from the cursor.

The table from which rows are deleted is determined as follows:

- If no FROM clause is included, the cursor must be on a single table only.
- If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.
- If a FROM clause is included, and no table owner is specified, *table-spec* is first matched against any correlation names.
  - If a correlation name exists, *table-spec* is identified with the correlation name.
  - If a correlation name does not exist, *table-spec* must be unambiguously identifiable as a table name in the cursor.
- If a FROM clause is included, and a table owner is specified, *table-spec* must be unambiguously identifiable as a table name in the cursor.
- The positioned DELETE statement can be used on a cursor open on a view as long as the view is updatable.

Permissions

Must have DELETE permission on tables used in the cursor.

Side effects

None.

See also

- "UPDATE statement" on page 703
- "UPDATE (positioned) statement [ESQL] [SP]" on page 708
- "INSERT statement" on page 573
- "PUT statement [ESQL]" on page 614

Standards and compatibility

- **SQL/2003** Core feature. The range of cursors that can be updated may contain vendor extensions if the ansi_update_constraints option is set to Off.

Example

The following statement removes the current row from the database.

```
DELETE
WHERE CURRENT OF cur_employee;
```


DESCRIBE statement [ESQL]

Use this statement to get information about the host variables required to store data retrieved from the database, or host variables required to pass data to the database.

Syntax

DESCRIBE
[**USER TYPES**]
[**ALL** | **BIND VARIABLES FOR** | **INPUT** | **OUTPUT**
| **SELECT LIST FOR**]
[**LONG NAMES** [*long-name-spec*] | **WITH VARIABLE RESULT**]
[**FOR**] { *statement-name* | **CURSOR** *cursor-name* }
INTO *sqlda-name*

long-name-spec :
OWNER.TABLE.COLUMN | **TABLE.COLUMN** | **COLUMN**

statement-name : *identifier* | *hostvar*

cursor-name : *declared cursor*

sqlda-name : *identifier*

Parameters

USER TYPES A DESCRIBE statement with the USER TYPES clause returns information about domains of a column. Typically, such a DESCRIBE is done when a previous DESCRIBE returns an indicator of DT_HAS_USERTYPE_INFO.

The information returned is the same as for a DESCRIBE without the USER TYPES keywords, except that the sqlname field holds the name of the domain, instead of the name of the column.

If the DESCRIBE uses the LONG NAMES clause, the sqldata field holds this information.

ALL DESCRIBE ALL allows you to describe INPUT and OUTPUT with one request to the database server. This has a performance benefit. The INPUT information is filled in the SQLDA first, followed by the OUTPUT information. The sqld field contains the total number of INPUT and OUTPUT variables. The DT_DESCRIBE_INPUT bit in the indicator variable is set for INPUT variables and clear for OUTPUT variables.

INPUT A bind variable is a value supplied by the application when the database executes the statements. Bind variables can be considered parameters to the statement. DESCRIBE INPUT fills in the name fields in the SQLDA with the bind variable names. DESCRIBE INPUT also puts the number of bind variables in the sqlda field of the SQLDA.

DESCRIBE uses the indicator variables in the SQLDA to provide additional information. DT_PROCEDURE_IN and DT_PROCEDURE_OUT are bits that are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns will have both bits clear. After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE). DESCRIBE INPUT only sets DT_PROCEDURE_IN and DT_PROCEDURE_OUT

appropriately when a bind variable is an argument to a CALL statement; bind variables within an expression that is an argument in a CALL statement will not set the bits.

OUTPUT The DESCRIBE OUTPUT statement fills in the data type and length for each select list item in the SQLDA. The name field is also filled in with a name for the select list item. If an alias is specified for a select list item, the name will be that alias. Otherwise, the name is derived from the select list item: if the item is a simple column name, it is used; otherwise, a substring of the expression is used. DESCRIBE will also put the number of select list items in the sqld field of the SQLDA.

If the statement being described is a UNION of two or more SELECT statements, the column names returned for DESCRIBE OUTPUT are the same column names which would be returned for the first SELECT statement.

If you describe a CALL statement, the DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each INOUT or OUT parameter in the procedure. DESCRIBE OUTPUT also puts the number of INOUT or OUT parameters in the sqld field of the SQLDA.

If you describe a CALL statement with a result set, the DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each RESULT column in the procedure definition. DESCRIBE OUTPUT will also put the number of result columns in the sqld field of the SQLDA.

LONG NAMES The LONG NAMES clause is provided to retrieve column names for a statement or cursor. Without this clause, there is a 29-character limit on the length of column names; with the clause, names of an arbitrary length are supported.

If LONG NAMES is used, the long names are placed into the SQLDATA field of the SQLDA, as if you were fetching from a cursor. None of the other fields (SQLLEN, SQLTYPE, and so on) are filled in. The SQLDA must be set up like a FETCH SQLDA: it must contain one entry for each column, and the entry must be a string type. If there is an indicator variable, truncation is indicated in the usual fashion.

The default specification for the long names is **TABLE.COLUMN**.

WITH VARIABLE RESULT This clause is used to describe procedures that can have more than one result set, with different numbers or types of columns.

If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the DESCRIBE statement to one of the following values:

- **0** The result set may change. The procedure call should be described again following each OPEN statement.
- **1** The result set is fixed. No redescribing is required.

For more information on the use of the SQLDA structure, see "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*].

Remarks

The DESCRIBE statement sets up the named SQLDA to describe either the OUTPUT (equivalently SELECT LIST) or the INPUT (BIND VARIABLES) for the named statement.

In the INPUT case, DESCRIBE BIND VARIABLES does not set up the data types in the SQLDA: this needs to be done by the application. The ALL keyword allows you to describe INPUT and OUTPUT in one SQLDA.

If you specify a statement name, the statement must have been previously prepared using the PREPARE statement with the same statement name and the SQLDA must have been previously allocated (see the "ALLOCATE DESCRIPTOR statement [ESQL]" on page 299).

If you specify a cursor name, the cursor must have been previously declared and opened. The default action is to describe the OUTPUT. Only SELECT statements and CALL statements have OUTPUT. A DESCRIBE OUTPUT on any other statement, or on a cursor that is not a dynamic cursor, indicates no output by setting the sqld field of the SQLDA to zero.

In embedded SQL, NCHAR, NVARCHAR and LONG NVARCHAR are described as DT_FIXCHAR, DT_VARCHAR, and DT_LONGVARCHAR, respectively, by default. If the db_change_nchar_charset function has been called, these data types are described as DT_NFIXCHAR, DT_NVARCHAR and DT_LONGNVARCHAR, respectively. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

For more information on how NCHAR data types are described, see the documentation for the data type: "NCHAR data type" on page 50, "NVARCHAR data type" on page 52, and "LONG NVARCHAR data type" on page 49.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 299
- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "OPEN statement [ESQL] [SP]" on page 601
- "PREPARE statement [ESQL]" on page 610

Standards and compatibility

- **SQL/2003** Core feature. Some clauses are vendor extensions.

Example

The following example shows how to use the DESCRIBE statement:

```
sqlda = alloc_sqlda( 3 );
EXEC SQL DESCRIBE OUTPUT
  FOR employee_statement
  INTO sqlda;
if( sqlda->sqld  >  sqlda->sqln ) {
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE OUTPUT
    FOR employee_statement
```



```
        INTO sqlda;
    }
```

DESCRIBE statement [Interactive SQL]

The DESCRIBE statement enables you to obtain all columns found in a table or view, all indexes found in a table, and all parameters used with a stored procedure or a function.

Syntax

DESCRIBE [[**INDEX FOR**] **TABLE** | **PROCEDURE**] [*owner.*]*object-name*

Parameters

INDEX FOR Indicates that you want to see the indexes for the specified *object-name*.

TABLE Indicates that the object to be described is a table or a view.

PROCEDURE Indicates that the object to be described is a procedure or a function.

Remarks

Use DESCRIBE TABLE to list all of the columns in the specified table or view. The DESCRIBE TABLE statement returns one row per table column. The output is formatted into four columns:

- **Column** The name of the column to describe.
- **Type** The type of data in the column.
- **Nullable** Whether nulls are allowed (1=yes, 0=no).
- **Primary Key** Whether the column is in the primary key (1=yes, 0=no).

Use DESCRIBE INDEX FOR TABLE to list all of the indexes for the specified table. The DESCRIBE TABLE statement returns one row per index. The output is formatted into four columns:

- **Index Name** The name of the index.
- **Columns** The columns in the index.
- **Unique** Whether the index is unique (1=yes, 0=no).
- **Type** The type of index. Possible choices are: Clustered, Statistic, Hashed, and Other.

Use DESCRIBE PROCEDURE to list all of the parameters used by the specified procedure or function. The DESCRIBE PROCEDURE statement returns one row for each parameter. The output is formatted into three columns:

- **Parameter** The name of the parameter.
- **Type** The data type of the parameter.
- **In/Out** Information about what is passed to, or returned from the procedure or function:
  - In - the caller passes data to the procedure, but does not receive data back
  - Out - the caller does not pass data to the procedure or function, but receives data back

- In/Out - the caller passes data to the procedure and receives data back
- Result - the procedure or function returns a result set
- Return - the procedure or function returns a declared return value

If you do not specify either TABLE or PROCEDURE (for example, DESCRIBE *object-name*) Interactive SQL assumes the object is a table. However, if no such table exists, Interactive SQL attempts to describe the object as either a procedure or a function.

Permissions

None

Side effects

None

See also

None

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

List the columns in the Departments table:

```
DESCRIBE TABLE Departments;
```

Here is an example of the result set for this statement:

Column	Type	Nullable	Primary key
DepartmentID	integer	0	1
DepartmentName	char(40)	0	0
DepartmentHeadID	integer	0	0

List the indexes for the Customers table:

```
DESCRIBE INDEX FOR TABLE Customers;
```

Here is an example of the results for this statement:

Index Name	Columns	Unique	Type
IX_customer_name	Surname,GivenName	0	Clustered

DETACH TRACING statement

Use this statement to end a diagnostic tracing session.

Syntax

DETACH TRACING { **WITH** | **WITHOUT** } **SAVE**

Parameters

WITH SAVE Specify WITH SAVE to save data any unsaved diagnostic data in the diagnostic tables.

WITHOUT SAVE Specify WITHOUT SAVE if you do not want to save any unsaved tracing data.

Remarks

Issue this statement from the database being profiled to stop sending diagnostic information to the diagnostic tables. If you specify the WITHOUT SAVE clause, you can still save the data later—assuming the tracing database is still running and another tracing session has not been started—by using the sa_save_trace_data system procedure. See "sa_save_trace_data system procedure" on page 912.

To see the current tracing levels set for a database, look in the sa_diagnostic_tracing_level table. See "sa_diagnostic_tracing_level table" on page 748.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ATTACH TRACING statement" on page 344
- "REFRESH TRACING LEVEL statement" on page 623
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]
- "sa_diagnostic_tracing_level table" on page 748
- "sa_save_trace_data system procedure" on page 912

Standards and compatibility

- **SQL/2003** Vendor extension.

DISCONNECT statement [ESQL] [Interactive SQL]

Use this statement to drop the current connection to a database.

Syntax

DISCONNECT [*connection-name* | **CURRENT** | **ALL**]

connection-name : *identifier*, *string*, or *hostvar*

Remarks

The DISCONNECT statement drops a connection with the database server and releases all resources used by it. If the connection to be dropped was named on the CONNECT statement, the name can be specified. Specifying ALL will drop all of the application's connections to all database environments. CURRENT is the default, and will drop the current connection.

Before closing the database connection, Interactive SQL automatically executes a COMMIT statement if the commit_on_exit option is set to On. If this option is set to Off, Interactive SQL performs an implicit ROLLBACK. By default, the commit_on_exit option is set to On.

For information on dropping connections other than the current connection, see "DROP CONNECTION statement" on page 500.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 370
- "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 661

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Example

The following statement shows how to use DISCONNECT in embedded SQL:

```
EXEC SQL DISCONNECT :conn_name
```

The following statement shows how to use DISCONNECT from Interactive SQL to disconnect all connections:

```
DISCONNECT ALL;
```


DROP statement

Use this statement to remove objects from the database.

Syntax

DROP
{ **DOMAIN** | **DATATYPE** } *datatype-name*
| **DBSPACE** *dbspace-name*
| **EVENT** *event-name*
| **FUNCTION** [*owner*.]*function-name*
| **INDEX** { [[*owner*.]*table-name*.]*index-name* | [[*owner*.]*materialized-view-name*.]*index-name* }
| **MESSAGE** *msgnum*
| **PROCEDURE** [*owner*.]*procedure-name*
| **TABLE** [*owner*.]*table-name*
| **TRIGGER** [[*owner*.]*table-name*.]*trigger-name*
| **VIEW** [*owner*.]*view-name*
| **MATERIALIZED VIEW** [*owner*.]*materialized-view-name*

Remarks

The DROP statement removes the definition of the indicated database object. If the object is a dbspace, all tables in that dbspace must be dropped prior to dropping the dbspace. If the object is a table or materialized view, all data in the table is automatically deleted as part of the dropping process. Also, all indexes and keys for a table or materialized view are dropped as well. You cannot use the DROP DBSPACE statement to drop the pre-defined dbspaces SYSTEM, TEMPORARY, TEMP, TRANSLOG, or TRANSLOGMIRROR. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

DROP TABLE, DROP MATERIALIZED VIEW, DROP INDEX, DROP DBSPACE, DROP PROCEDURE and DROP FUNCTION are prevented whenever the statement affects an object that is currently being used by another connection. DROP TABLE is prevented if there is a materialized view dependent on the table.

DROP TABLE, DROP MATERIALIZED VIEW, and DROP VIEW cause the status of all dependent non-materialized views to become INVALID. To determine view dependencies before dropping a table, view or materialized view, use the sa_dependent_views system procedure. See "sa_dependent_views system procedure" on page 859.

DROP DOMAIN is prevented if the data type is used in a table column, or in a procedure or function argument. You must change data types on all columns defined using the domain in order to drop the data type. It is recommended that you use DROP DOMAIN rather than DROP DATATYPE, as DROP DOMAIN is the syntax used in the ANSI/ISO SQL3 draft. You cannot drop system-defined data types (such as MONEY or UNIQUEIDENTIFIERSTR) from a database.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP statement.

For DROP DBSPACE, you must be the only connection to the database.

A user with ALTER permissions on the table can execute DROP TRIGGER.

A user with REFERENCES permissions on the table can execute DROP INDEX.

Global temporary tables cannot be dropped unless all users that have referenced the temporary table have disconnected.

The DROP INDEX statement cannot be used within a snapshot transaction when snapshot isolation is enabled for the database. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit. Clears the Results tab in the Results pane in Interactive SQL. DROP TABLE, DROP VIEW, DROP MATERIALIZED VIEW, and DROP INDEX close all cursors for the current connection.

DROP TABLE can be used to drop a local temporary table, but DROP INDEX cannot be used to drop an index on a local temporary table. An attempt to do so results in an Index not found error. Indexes on local temporary tables are dropped automatically when the local temporary table goes out of scope.

When a view is dropped, all procedures and triggers are unloaded from memory, so that any procedure or trigger that references the view reflects the fact that the view does not exist. The unloading and loading of procedures and triggers can have a performance impact if you are regularly dropping and creating views.

See also

- "CREATE DATABASE statement" on page 374
- "CREATE DOMAIN statement" on page 386
- "CREATE INDEX statement" on page 405
- "CREATE FUNCTION statement" on page 399
- "CREATE PROCEDURE statement" on page 414
- "CREATE TABLE statement" on page 450
- "CREATE TRIGGER statement" on page 462
- "CREATE VIEW statement" on page 471
- "CREATE STATISTICS statement" on page 442
- "CREATE MATERIALIZED VIEW statement" on page 411

Standards and compatibility

- **SQL/2003** Core feature, however the support for materialized views is a vendor extension.

Example

Drop the Departments table from the database.

```
DROP TABLE Departments;
```

Drop the EmployeesAndDepartments view from the database.

```
DROP VIEW EmployeesAndDepartments;
```

Drop the price index from the ProductIDsPerCustomer materialized view.

```
DROP INDEX ProductIDsPerCustomer.price;
```


DROP CONNECTION statement

Use this statement to drop a user's connection to the database.

Syntax

DROP CONNECTION *connection-id*

Remarks

The DROP CONNECTION statement disconnects a user from the database by dropping the connection to the database.

The *connection-id* parameter is an integer constant. You can obtain the *connection-id* using the sa_conn_info system procedure.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 370
- "sa_conn_info system procedure" on page 850

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following procedure drops a connection identified by its connection number. Note that when executing the DROP CONNECTION statement from within a procedure, you should do so using the EXECUTE IMMEDIATE statement, as shown in this example:

```
CREATE PROCEDURE drop_connection_by_id( IN conn_number INTEGER )
 BEGIN
   EXECUTE IMMEDIATE 'DROP CONNECTION ' || conn_number;
 END;
```

The following statement drops the connection with ID number 4.

```
DROP CONNECTION 4;
```


DROP DATABASE statement

Use this statement to delete all database files associated with a database.

Syntax

DROP DATABASE *database-name* [**KEY** *key*]

Remarks

The DROP DATABASE statement physically deletes all associated database files from disk. If the database file does not exist, or is not in a suitable condition for the database to be started, an error is generated.

DROP DATABASE cannot be used in stored procedures, triggers, events, or batches.

Permissions

Required permissions are set using the database server -gu option. The default setting is to require DBA authority.

The database must not be in use to be dropped.

You must specify a key if you want to drop a strongly encrypted database

Not supported on Windows CE.

Side effects

In addition to deleting the database files from disk, any associated transaction log file or transaction log mirror file is deleted.

See also

- "CREATE DATABASE statement" on page 374
- "DatabaseKey connection parameter [DBKEY]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Drop the database *temp.db*, in the *C:\temp* directory:

```
DROP DATABASE 'c:\temp\temp.db';
```


DROP EXTERNLOGIN statement

Use this statement to drop an external login from the SQL Anywhere catalogs.

Syntax

DROP EXTERNLOGIN *login-name* **TO** *remote-server*

Parameters

DROP clause Specifies the local user login name

TO clause Specifies the name of the remote server. The local user's alternate login name and password for that server is the external login that is deleted.

Remarks

DROP EXTERNLOGIN deletes an external login from the SQL Anywhere catalogs.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE EXTERNLOGIN statement" on page 397

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
DROP EXTERNLOGIN DBA TO sybase1;
```


DROP PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to drop a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax

DROP PUBLICATION [*owner*.]*publication-name*

owner, *publication-name* : *identifier*

Remarks

This statement is applicable only to MobiLink and SQL Remote.

Permissions

Must have DBA authority.

Side effects

Automatic commit. All subscriptions to the publication are dropped.

See also

- "ALTER PUBLICATION statement [MobiLink] [SQL Remote]" on page 317
- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 427
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "UltraLite DROP PUBLICATION statement" [*UltraLite - Database Management and Reference*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement drops the pub_contact publication.

```
DROP PUBLICATION pub_contact;
```


DROP REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to delete a message type definition from a database.

Syntax

DROP REMOTE MESSAGE TYPE *message-system*

message-system: **FILE** | **FTP** | **MAPI** | **SMTP** | **VIM**

Remarks

The statement removes a message type from a database.

> **Note**
> Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority. To be able to drop the type, there must be no user granted REMOTE or CONSOLIDATE permissions with this type.

Side effects

Automatic commit.

See also

- "CREATE REMOTE MESSAGE TYPE statement [SQL Remote]" on page 431
- "Using message types" [*SQL Remote*].

Example

The following statement drops the FILE message type from a database.

```
DROP REMOTE MESSAGE TYPE file;
```

DROP SERVER statement

Use this statement to drop a remote server from the SQL Anywhere catalog.

Syntax

DROP SERVER *server-name*

Remarks

DROP SERVER deletes a remote server from the SQL Anywhere catalogs. You must drop all the proxy tables that have been defined for the remote server before this statement will succeed.

Permissions

Only user DBA can delete a remote server.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "CREATE SERVER statement" on page 435

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
DROP SERVER ase_prod;
```


DROP SERVICE statement

Use this statement to drop a web service.

Syntax

DROP SERVICE *service-name*

Remarks

This statement deletes a web service listed in the ISYSWEBSERVICE system table.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER SERVICE statement" on page 323
- "CREATE SERVICE statement" on page 438
- "ISYSWEBSERVICE system table" on page 734

Example

To drop a web service named tables, execute the following statement:

```
DROP SERVICE tables;
```

DROP STATEMENT statement [ESQL]

Use this statement to free statement resources.

Syntax

DROP STATEMENT [*owner*.]*statement-name*

statement-name : *identifier* | *hostvar*

Remarks

The DROP STATEMENT statement frees resources used by the named prepared statement. These resources are allocated by a successful PREPARE statement, and are normally not freed until the database connection is released.

Permissions

Must have prepared the statement.

Side effects

None.

See also

- "PREPARE statement [ESQL]" on page 610

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following are examples of DROP STATEMENT use:

```
EXEC SQL DROP STATEMENT S1;
EXEC SQL DROP STATEMENT :stmt;
```


DROP STATISTICS statement

Use this statement to erase all column statistics on the specified columns.

Syntax

DROP STATISTICS [**ON**] [*owner.*]*table-name* [**(** *column-list* **)**]

Remarks

The SQL Anywhere optimizer uses column statistics to determine the best strategy for executing each statement. SQL Anywhere automatically gathers and updates these statistics. Column statistics are stored permanently in the database in the ISYSCOLSTAT system table. Column statistics gathered while processing one statement are available when searching for efficient ways to execute subsequent statements.

Occasionally, the column statistics can become inaccurate or relevant statistics may be unavailable. This condition is most likely to arise when few queries have been executed since a large amount of data was added, updated, or deleted.

The DROP STATISTICS statement deletes all internal statistical data from the ISYSCOLSTAT system table for the specified columns. This drastic step leaves the optimizer with no access to essential statistical information. Without these statistics, the optimizer can generate very inefficient data access plans, causing poor database performance.

The DROP STATISTICS statement requires an exclusive lock on the table against which it is being performed. This means that execution of the statement cannot proceed until all other connections that refer to the table have either committed or rolled back the referring transactions, or closed any open cursors that refer to the table.

This statement should be used only during problem determination or when reloading data into a database that differs substantially from the original data.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE STATISTICS statement" on page 442
- "ISYSCOLSTAT system table" on page 727

Standards and compatibility

- **SQL/2003** Vendor extension.

DROP SUBSCRIPTION statement [SQL Remote]

Use this statement to drop a subscription for a user from a publication.

Syntax

DROP SUBSCRIPTION TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*, …

subscription-value: *string*

subscriber-id: *string*

Parameters

publication-name The name of the publication to which the user is being subscribed. This can include the owner of the publication.

subscription-value A string that is compared to the subscription expression of the publication. This value is required because a user can have more than one subscription to a publication.

subscriber-id The user ID of the subscriber to the publication.

Remarks

Drops a SQL Remote subscription for a user ID to a publication in the current database. The user ID will no longer receive updates when data in the publication is changed.

In SQL Remote, publications and subscriptions are two-way relationships. If you drop a subscription for a remote user to a publication on a consolidated database, you should also drop the subscription for the consolidated database on the remote database to prevent updates on the remote database being sent to the consolidated database.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 443
- "ISYSSUBSCRIPTION system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement drops a subscription for the SamS user ID to the pub_contact publication.

```
DROP SUBSCRIPTION TO pub_contact
FOR SamS;
```


DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement to drop a synchronization subscription in a MobiLink remote database.

Syntax

DROP SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, …]

Parameters

TO clause Specify the name of a publication.

FOR clause Specify one more MobiLink users.

Omitting this clause drops the default settings for the publication.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side Effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 328
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 445
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example drops the subscription between the MobiLink user ml_user1 and the publication called sales_publication:

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR "ml_user1";
```

The following example omits the FOR clause, and so drops the default settings for the publication called sales_publication:

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication;
```


DROP SYNCHRONIZATION USER statement [MobiLink]

Use this statement to drop one or more synchronization users from a SQL Anywhere remote database.

Syntax

DROP SYNCHRONIZATION USER *ml_username*, …

ml_username: *identifier*

Remarks

Drop one or more synchronization users from a MobiLink remote database.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side Effects

All subscriptions associated with the user are also deleted.

See also

- "ALTER SYNCHRONIZATION USER statement [MobiLink]" on page 330
- "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 448
- "ISYSSYNC system table" on page 732

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Remove MobiLink user ml_user1 from the database.

```
DROP SYNCHRONIZATION USER ml_user1;
```


DROP VARIABLE statement

Use this statement to eliminate a SQL variable.

Syntax

DROP VARIABLE *identifier*

Remarks

The DROP VARIABLE statement eliminates a SQL variable that was previously created using the CREATE VARIABLE statement. Variables are automatically eliminated when the database connection is released. Variables are often used for large objects, so eliminating them after use or setting them to NULL can free up significant resources (primarily disk space).

Permissions

None.

Side effects

None.

See also

- "CREATE VARIABLE statement" on page 469
- "SET statement" on page 656

Standards and compatibility

- **SQL/2003** Vendor extension.

EXCEPT statement

Computes the difference between the result sets of two or more queries.

Syntax

[**WITH** *temporary-views*] *query-block*
EXCEPT [**ALL** | **DISTINCT**] *query-block*
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], …]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

Note
You cannot use the FOR, FOR XML, WITH, or OPTION clause in the *query-block*.

OPTION clause

This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].

- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The differences between the result sets of several query blocks can be obtained as a single result using EXCEPT or EXCEPT ALL. EXCEPT DISTINCT is identical to EXCEPT.

The *query-block* must each have the same number of items in the select list.

The number of rows in the result set of EXCEPT ALL is exactly the difference between the number of rows in the result sets of the separate queries.

The results of EXCEPT are the same as EXCEPT ALL, except that when using EXCEPT, duplicate rows are eliminated before the difference between the result sets is computed.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately. The first query specification of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None

See also

- "INTERSECT statement" on page 580
- "UNION statement" on page 695

Standards and compatibility

- **SQL/2003** EXCEPT DISTINCT is a core feature. EXCEPT ALL is feature F304.

Example

For examples of EXCEPT usage, see "Set operators and NULL" [*SQL Anywhere Server - SQL Usage*].

EXECUTE statement [ESQL]

Use this statement to execute a prepared SQL statement.

Syntax 1

EXECUTE *statement*
[**USING** { *hostvar-list* | **DESCRIPTOR** *sqlda-name* }]
[**INTO** { *into-hostvar-list* | **DESCRIPTOR** *into-sqlda-name* }]
[**ARRAY :***integer*]

statement : *identifier* | *hostvar* | *string*

sqlda-name : *identifier*

into-sqlda-name : *identifier*

Syntax 2

EXECUTE IMMEDIATE *statement*

statement : *string* | *hostvar*

Parameters

USING clause Results from a SELECT statement or a CALL statement are put into either the variables in the variable list or the program data areas described by the named SQLDA. The correspondence is one-to-one from the OUTPUT (selection list or parameters) to either the host variable list or the SQLDA descriptor array.

INTO clause If EXECUTE INTO is used with an INSERT statement, the inserted row is returned in the second descriptor. For example, when using auto-increment primary keys or BEFORE INSERT triggers that generate primary key values, the EXECUTE statement provides a mechanism to re-fetch the row immediately and determine the primary key value that was assigned to the row. The same thing can be achieved by using @@identity with auto-increment keys.

ARRAY clause The optional ARRAY clause can be used with prepared INSERT statements to allow wide inserts, which insert more than one row at a time and which can improve performance. The integer value is the number of rows to be inserted. The SQLDA must contain a variable for each entry (number of rows * number of columns). The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

Remarks

The EXECUTE statement can be used for any SQL statement that can be prepared. Cursors are used for SELECT statements or CALL statements that return many rows from the database (see "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*]).

After successful execution of an INSERT, UPDATE or DELETE statement, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) is filled in with the number of rows affected by the operation.

Syntax 1 Execute the named dynamic statement, which was previously prepared. If the dynamic statement contains host variable place holders which supply information for the request (bind variables), either the

sqlda-name must specify a C variable which is a pointer to a SQLDA containing enough descriptors for all of the bind variables occurring in the statement, or the bind variables must be supplied in the *hostvar -list*.

Syntax 2 A short form to PREPARE and EXECUTE a statement that does not contain bind variables or output. The SQL statement contained in the string or host variable is immediately executed, and is dropped on completion.

Permissions

Permissions are checked on the statement being executed.

Side effects

None.

See also

- "EXECUTE IMMEDIATE statement [SP]" on page 519
- "PREPARE statement [ESQL]" on page 610
- "DECLARE CURSOR statement [ESQL] [SP]" on page 478

Standards and compatibility

- **SQL/2003** Feature outside of core SQL.

Example

Execute a DELETE.

```
EXEC SQL EXECUTE IMMEDIATE
'DELETE FROM Employees WHERE EmployeeID = 105';
```

Execute a prepared DELETE statement.

```
EXEC SQL PREPARE del_stmt FROM
'DELETE FROM Employees WHERE EmployeeID = :a';
EXEC SQL EXECUTE del_stmt USING :employee_number;
```

Execute a prepared query.

```
EXEC SQL PREPARE sel1 FROM
'SELECT Surname FROM Employees WHERE EmployeeID = :a';
EXEC SQL EXECUTE sel1 USING :employee_number INTO :surname;
```


EXECUTE statement [T-SQL]

Use Syntax 1 to invoke a procedure, as an Adaptive Server Enterprise-compatible alternative to the CALL statement. Use Syntax 2 to execute a prepared SQL statement in Transact-SQL.

Syntax 1

EXECUTE [*@return_status* =] [*creator*.]*procedure_name* [*argument*, …]

argument :
[*@parameter-name* =] *expression*
| [*@parameter-name* =] *@variable* [*output*]

Syntax 2

EXECUTE (*string-expression* **)**

Remarks

Syntax 1 executes a stored procedure, optionally supplying procedure parameters and retrieving output values and return status information.

The EXECUTE statement is implemented for Transact-SQL compatibility, but can be used in either Transact-SQL or Watcom-SQL batches and procedures.

With Syntax 2, you can execute statements within Transact-SQL stored procedures and triggers. The EXECUTE statement extends the range of statements that can be executed from within procedures and triggers. It lets you execute dynamically prepared statements, such as statements that are constructed using the parameters passed in to a procedure. Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

The Transact-SQL EXECUTE statement does not have a way to signify that a result set is expected. One way to indicate that a Transact-SQL procedure returns a result set is to include something like the following:

```
IF 1 = 0 THEN
      SELECT 1 AS a
```

You can also execute statements within Transact-SQL stored procedures and triggers. See “EXECUTE IMMEDIATE statement [SP]” on page 519.

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

- “CALL statement” on page 357
- “EXECUTE statement [ESQL]” on page 515
- “EXECUTE IMMEDIATE statement [SP]” on page 519

Example

The following procedure illustrates Syntax 1.


```
CREATE PROCEDURE p1( @var INTEGER = 54 )
AS
PRINT 'on input @var = %1!', @var
DECLARE @intvar integer
SELECT @intvar=123
SELECT @var=@intvar
PRINT 'on exit @var = %1!', @var;
```

The following statement executes the procedure, supplying the input value of 23 for the parameter. If you are connected from an Open Client or JDBC application, the PRINT messages are displayed on the client window. If you are connected from an ODBC or embedded SQL application, the messages are displayed on the Server Messages window.

```
EXECUTE p1 23;
```

The following is an alternative way of executing the procedure, which is useful if there are several parameters.

```
EXECUTE p1 @var = 23;
```

The following statement executes the procedure, using the default value for the parameter

```
EXECUTE p1;
```

The following statement executes the procedure, and stores the return value in a variable for checking return status.

```
EXECUTE @status = p1 23;
```


EXECUTE IMMEDIATE statement [SP]

Use this statement to enable dynamically-constructed statements to be executed from within a procedure.

Syntax 1

EXECUTE IMMEDIATE [*execute-option*] *string-expression*

execute-option:
WITH QUOTES [**ON** | **OFF**]
| **WITH ESCAPES** { **ON** | **OFF** }
| **WITH RESULT SET** { **ON** | **OFF** }

Syntax 2

EXECUTE (*string-expression* **)**

Parameters

WITH QUOTES When you specify WITH QUOTES or WITH QUOTES ON, any double quotes in the string expression are assumed to delimit an identifier. When you do not specify WITH QUOTES, or specify WITH QUOTES OFF, the treatment of double quotes in the string expression depends on the current setting of the quoted_identifier option.

WITH QUOTES is useful when an object name that is passed into the stored procedure is used to construct the statement that is to be executed, but the name might require double quotes and the procedure might be called when the quoted_identifier option is set to Off. See "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

WITH ESCAPES WITH ESCAPES OFF causes any escape sequences (such as \n, \x, or \\) in the string expression to be ignored. For example, two consecutive backslashes remain as two backslashes, rather than being converted to a single backslash. The default setting is equivalent to WITH ESCAPES ON.

One use of WITH ESCAPES OFF is for easier execution of dynamically-constructed statements referencing file names that contain backslashes.

In some contexts, escape sequences in the *string-expression* are transformed before the EXECUTE IMMEDIATE statement is executed. For example, compound statements are parsed before being executed, and escape sequences are transformed during this parsing, regardless of the WITH ESCAPES setting. In these contexts, WITH ESCAPES OFF prevents further translations from occurring. For example:

```
BEGIN
   DECLARE String1 LONG VARCHAR;
   DECLARE String2 LONG VARCHAR;
   EXECUTE IMMEDIATE
        'SET String1 = ''One backslash: \\\\ ''';
     EXECUTE IMMEDIATE WITH ESCAPES OFF
        'SET String2 = ''Two backslashes: \\\\ ''';
   SELECT String1, String2
END
```

WITH RESULT SET You can have an EXECUTE IMMEDIATE statement return a result set by specifying WITH RESULT SET ON. With this clause, the containing procedure is marked as returning a result set. If you do not include this clause, an error is reported when the procedure is called if the statement produces a result set.

Note
The default option is WITH RESULT SET OFF, meaning that no result set is produced when the statement is executed.

Remarks

The EXECUTE statement extends the range of statements that can be executed from within procedures and triggers. It lets you execute dynamically-prepared statements, such as statements that are constructed using the parameters passed in to a procedure.

Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

Only global variables can be referenced in a statement executed by EXECUTE IMMEDIATE.

Only syntax 2 can be used inside Transact-SQL stored procedures and triggers.

Permissions

None. The statement is executed with the permissions of the owner of the procedure, not with the permissions of the user who calls the procedure.

Side effects

None. However, if the statement is a data definition statement with an automatic commit as a side effect, that commit does take place.

For more information about using the EXECUTE IMMEDIATE statement in procedures, see "Using the EXECUTE IMMEDIATE statement in procedures" [*SQL Anywhere Server - SQL Usage*].

See also

- "CREATE PROCEDURE statement" on page 414
- "BEGIN statement" on page 351
- "EXECUTE statement [ESQL]" on page 515

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Examples

The following procedure creates a table, where the table name is supplied as a parameter to the procedure. The EXECUTE IMMEDIATE statement must all be on a single line.

```
CREATE PROCEDURE CreateTableProc(
                  IN tablename char(30)
                  )
BEGIN
   EXECUTE IMMEDIATE
   'CREATE TABLE ' || tablename ||
   ' ( column1 INT PRIMARY KEY)'
END;
```

To call the procedure and create a table called mytable:

```
CALL CreateTableProc( 'mytable' );
```


For an example of EXECUTE IMMEDIATE with a query that returns a result set, see "Using the EXECUTE IMMEDIATE statement in procedures" [*SQL Anywhere Server - SQL Usage*].

EXIT statement [Interactive SQL]

Use this statement to leave Interactive SQL.

Syntax

{ **EXIT** | **QUIT** | **BYE** } [*return-code*]

return-code: *number* | *connection-variable*

Remarks

This statement closes the Interactive SQL window if you are running Interactive SQL as a windowed program, or terminates Interactive SQL altogether when run in command-prompt (batch) mode. In both cases, the database connection is also closed. Before closing the database connection, Interactive SQL automatically executes a COMMIT statement if the commit_on_exit option is set to On. If this option is set to Off, Interactive SQL performs an implicit ROLLBACK. By default, the commit_on_exit option is set to On.

The optional return code can be used in batch files to indicate success or failure of the commands in an Interactive SQL command file. The default return code is 0.

Permissions

None.

Side effects

This statement automatically performs a commit if option commit_on_exit is set to On (the default); otherwise it performs an implicit rollback.

On Windows operating systems the optional return value is available as ERRORLEVEL.

See also

- "SET OPTION statement" on page 664

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example sets the Interactive SQL return value to 1 if there are any rows in table T, or to 0 if T contains no rows.

```
CREATE VARIABLE rowCount INT;
CREATE VARIABLE retcode INT;
SELECT COUNT(*) INTO rowCount FROM T;
IF( rowCount > 0 ) THEN
    SET retcode = 1;
ELSE
    SET retcode = 0;
END IF;
EXIT retcode;
```


Note
You cannot write the following the statement because EXIT is an Interactive SQL statement (not a SQL statement), and you cannot include any Interactive SQL statement in other SQL block statements.

```
CREATE VARIABLE rowCount INT;
SELECT COUNT(*) INTO rowCount FROM T;
IF( rowCount > 0 ) THEN
    EXIT 1;    //  <-- not allowed
ELSE
    EXIT 0;    //  <-- not allowed
END IF;
```

The following Windows batch file prints `Error = 1` on the command prompt.

```
dbisql -c "DSN=SQL Anywhere 10 Demo" EXIT 1
IF ERRORLEVEL 1 ECHO "Errorlevel is 1"
```

EXPLAIN statement [ESQL]

Use this statement to retrieve a text specification of the optimization strategy used for a particular cursor.

Syntax

EXPLAIN PLAN FOR CURSOR *cursor-name*
{ **INTO** *hostvar* | **USING DESCRIPTOR** *sqlda-name* }

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

Remarks

The EXPLAIN statement retrieves a text representation of the optimization strategy for the named cursor. The cursor must be previously declared and opened.

The *hostvar* or *sqlda-name* variable must be of string type. The optimization string specifies in what order the tables are searched, and also which indexes are being used for the searches if any.

This string may be long, depending on the query, and has the following format:

```
table (index), table (index), ...
```

If a table has been given a correlation name, the correlation name will appear instead of the table name. The order that the table names appear in the list is the order in which they are accessed by the database server. After each table is a parenthesized index name. This is the index that is used to access the table. If no index is used (the table is scanned sequentially) the letters "seq" will appear for the index name. If a particular SQL SELECT statement involves subqueries, a colon (:) will separate each subquery's optimization string. These subquery sections will appear in the order that the database server executes the queries.

After successful execution of the EXPLAIN statement, the sqlerrd field of the SQLCA (SQLIOESTIMATE) is filled in with an estimate of the number of input/output operations required to fetch all rows of the query.

A discussion with quite a few examples of the optimization string can be found in "Monitoring and Improving Performance" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have opened the named cursor.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "PREPARE statement [ESQL]" on page 610
- "FETCH statement [ESQL] [SP]" on page 526
- "CLOSE statement [ESQL] [SP]" on page 363
- "OPEN statement [ESQL] [SP]" on page 601
- "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*]
- "The SQL Communication Area (SQLCA)" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example illustrates the use of EXPLAIN:

```
EXEC SQL BEGIN DECLARE SECTION;
char plan[300];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE employee_cursor CURSOR FOR
   SELECT EmployeeID, Surname
   FROM Employees
   WHERE Surname like :pattern;
EXEC SQL OPEN employee_cursor;
EXEC SQL EXPLAIN PLAN FOR CURSOR employee_cursor INTO :plan;
printf( "Optimization Strategy: '%s'.n", plan );
```

The plan variable contains the following string:

```
'Employees <seq>'
```


FETCH statement [ESQL] [SP]

Use this statement to reposition a cursor and then get data from it.

Syntax

FETCH *cursor-position cursor-name*
[**INTO** { *hostvar-list* | *variable-list* } | **USING DESCRIPTOR** *sqlda-name*]
[**PURGE**]
[**BLOCK** *n*]
[**FOR UPDATE**]
[**ARRAY** *fetch-count*]
INTO *variable-list* [**FOR UPDATE**]

cursor-position :
  NEXT | **PRIOR** | **FIRST** | **LAST**
| { **ABSOLUTE** | **RELATIVE** } *row-count*

row-count : *number* or *hostvar*

cursor-name : *identifier* or *hostvar*

hostvar-list : may contain indicator variables

variable-list : stored procedure variables

sqlda-name : *identifier*

fetch-count : *integer* or *hostvar*

Parameters

INTO The INTO clause is optional. If it is not specified, the FETCH statement positions the cursor only. The *hostvar-list* is for embedded SQL use only.

cursor position An optional positional parameter allows the cursor to be moved before a row is fetched. If the fetch includes a positioning parameter and the position is outside the allowable cursor positions, the SQLE_NOTFOUND warning is issued and the SQLCOUNT field indicates the offset from a valid position.

The OPEN statement initially positions the cursor before the first row.

- **NEXT** Next is the default positioning, and causes the cursor to be advanced one row before the row is fetched.
- **PRIOR** Causes the cursor to be backed up one row before fetching.
- **RELATIVE** RELATIVE positioning is used to move the cursor by a specified number of rows in either direction before fetching. A positive number indicates moving forward and a negative number indicates moving backward. Thus, a NEXT is equivalent to RELATIVE 1 and PRIOR is equivalent to RELATIVE -1. RELATIVE 0 retrieves the same row as the last fetch statement on this cursor.
- **ABSOLUTE** The ABSOLUTE positioning parameter is used to go to a particular row. A zero indicates the position before the first row (see "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]).

A one (1) indicates the first row, and so on. Negative numbers are used to specify an absolute position from the end of the cursor. A negative one (-1) indicates the last row of the cursor.

- **FIRST** A short form for ABSOLUTE 1.
- **LAST** A short form for ABSOLUTE -1.

Cursor positioning problems
Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.
This occurs if a temporary table had to be created to open the cursor (see "Use work tables in query processing (use All-rows optimization goal)" [*SQL Anywhere Server - SQL Usage*] for a description).
The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

BLOCK clause Rows may be fetched by the client application more than one at a time. This is referred to as block fetching, prefetching, or multi-row fetching. The first fetch causes several rows to be sent back from the database server. The client buffers these rows, and subsequent fetches are retrieved from these buffers without a new request to the database server.

The BLOCK clause is for use in embedded SQL only. It gives the client and server a hint as to how many rows may be fetched by the application. The special value of 0 means the request is sent to the database server and a single row is returned (no row blocking). The BLOCK clause will reduce the number of rows included in the next prefetch to the BLOCK value. To increase the number of rows prefetched, use the PrefetchRows connection parameter.

If no BLOCK clause is specified, the value specified on OPEN is used. See "OPEN statement [ESQL] [SP]" on page 601.

FETCH RELATIVE 0 always re-fetches the row.

If prefetch is disabled for the cursor, the BLOCK clause is ignored and rows are fetched one at a time. If ARRAY is also specified, then the number of rows specified by ARRAY are fetched.

PURGE clause The PURGE clause is for use in embedded SQL only. It causes the client to flush its buffers of all rows, and then send the fetch request to the database server. Note that this fetch request may return a block of rows.

FOR UPDATE clause The FOR UPDATE clause indicates that the fetched row will subsequently be updated with an UPDATE WHERE CURRENT OF CURSOR statement. This clause causes the database server to put an intent lock on the row. The lock is held until the end of the current transaction. See "How locking works" [*SQL Anywhere Server - SQL Usage*] and the FOR UPDATE clause of the "SELECT statement" on page 648.

ARRAY clause The ARRAY clause is for use in embedded SQL only. It allows so-called wide fetches, which retrieve more than one row at a time, and which may improve performance.

To use wide fetches in embedded SQL, include the fetch statement in your code as follows:


```
EXEC SQL FETCH . . . ARRAY nnn
```

where ARRAY *nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

For a detailed example of using wide fetches, see "Fetching more than one row at a time" [*SQL Anywhere Server - Programming*].

Remarks

The FETCH statement retrieves one row from the named cursor. The cursor must have been previously opened.

Embedded SQL use A DECLARE CURSOR statement must appear before the FETCH statement in the C source code, and the OPEN statement must be executed before the FETCH statement. If a host variable is being used for the cursor name, the DECLARE statement actually generates code and thus must be executed before the FETCH statement.

The server returns in SQLCOUNT the number of records fetched, and always returns a SQLCOUNT greater than zero unless there is an error or warning. A SQLCOUNT of zero with no error condition indicates that one valid row has been fetched.

If the SQLSTATE_NOTFOUND warning is returned on the fetch, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) contains the number of rows by which the attempted fetch exceeded the allowable cursor positions. The value is 0 if the row was not found but the position is valid; for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value is positive if the attempted fetch was beyond the end of the cursor, and negative if the attempted fetch was before the beginning of the cursor. The cursor is positioned on the last row if the attempted fetch was beyond the end of the cursor, and on the first row if the attempted fetch was before the beginning of the cursor.

After successful execution of the fetch statement, the *sqlerrd*[*1*] field of the SQLCA (SQLIOCOUNT) is incremented by the number of input/output operations required to perform the fetch. This field is actually incremented on every database statement.

Single row fetch One row from the result of the SELECT statement is put into the variables in the variable list. The correspondence is one-to-one from the select list to the host variable list.

Multi-row fetch One or more rows from the result of the SELECT statement are put into either the variables in *variable-list* or the program data areas described by *sqlda-name*. In either case, the correspondence is one-to-one from the *select-list* to either the *hostvar-list* or the *sqlda-name* descriptor array.

Permissions

The cursor must be opened, and the user must have SELECT permission on the tables referenced in the declaration of the cursor.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478

- "PREPARE statement [ESQL]" on page 610
- "OPEN statement [ESQL] [SP]" on page 601
- "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*]
- "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]
- "FOR statement" on page 530
- "RESUME statement" on page 633 to retrieve multiple result sets

Standards and compatibility

- **SQL/2003** Core feature. Use in procedures is a Persistent Stored Module feature.

Example

The following is an embedded SQL example:

```
EXEC SQL DECLARE cur_employee CURSOR FOR
SELECT EmployeeID, Surname FROM Employees;
EXEC SQL OPEN cur_employee;
EXEC SQL FETCH cur_employee
INTO :emp_number, :emp_name:indicator;
```

The following is a procedure example:

```
BEGIN
   DECLARE cur_employee CURSOR FOR
      SELECT Surname
      FROM Employees;
   DECLARE name CHAR(40);
   OPEN cur_employee;
   lp: LOOP
      FETCH NEXT cur_employee into name;
      IF SQLCODE <> 0 THEN LEAVE lp END IF;
       ...
   END LOOP;
   CLOSE cur_employee;
END
```


FOR statement

Use this statement to repeat the execution of a statement list once for each row in a cursor.

Syntax

[*statement-label* :]
FOR *for-loop-name* **AS** *cursor-name* [*cursor-type*] **CURSOR**
{ **FOR** *statement* [**FOR** { **UPDATE** *cursor-concurrency* | **FOR READ ONLY** }]
| **USING** *variable-name* }
DO *statement-list*
END FOR [*statement-label*]

cursor-type :
NO SCROLL
| **DYNAMIC SCROLL**
| **SCROLL**
| **INSENSITIVE**
| **SENSITIVE**

cursor-concurrency : **BY** { **VALUES** | **TIMESTAMP** | **LOCK** }

variable-name : *identifier*

Parameters

NO SCROLL A cursor declared NO SCROLL is restricted to moving forward through the result set using FETCH NEXT and FETCH RELATIVE 0 seek operations.

As rows cannot be returned to once the cursor leaves the row, there are no sensitivity restrictions on the cursor. Consequently, when a NO SCROLL cursor is requested, SQL Anywhere supplies the most efficient kind of cursor, which is an asensitive cursor. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

DYNAMIC SCROLL DYNAMIC SCROLL is the default cursor type. DYNAMIC SCROLL cursors can use all formats of the FETCH statement.

When a DYNAMIC SCROLL cursor is requested, SQL Anywhere supplies an asensitive cursor. When using cursors there is always a trade-off between efficiency and consistency. Asensitive cursors provide efficient performance at the expense of consistency. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

SCROLL A cursor declared SCROLL can use all formats of the FETCH statement. When a SCROLL cursor is requested, SQL Anywhere supplies a value-sensitive cursor. See "Value-sensitive cursors" [*SQL Anywhere Server - Programming*].

SQL Anywhere must execute value-sensitive cursors in such a way that result set membership is guaranteed. DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required.

INSENSITIVE A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE

CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor. See "Insensitive cursors" [*SQL Anywhere Server - Programming*].

SENSITIVE A cursor declared SENSITIVE is sensitive to changes to membership or values of the result set. See "Sensitive cursors" [*SQL Anywhere Server - Programming*].

FOR UPDATE | READ ONLY A cursor declared FOR READ ONLY cannot be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default. Cursors default to FOR UPDATE for single-table queries without an ORDER BY clause, or if the ansi_update_constraints option is set to Off. When the ansi_update_constraints option is set to Cursors or Strict, then cursors over a query containing an ORDER BY clause default to READ ONLY. However, you can explicitly mark cursors as updatable using the FOR UPDATE clause. Because it is expensive to allow updates over cursors with an ORDER BY clause or a join, cursors over a query containing a join of two or more tables are READ ONLY and cannot be made updatable. In response to any request for a cursor that specifies FOR UPDATE, SQL Anywhere provides either a value-sensitive cursor or a sensitive cursor. Insensitive and asensitive cursors are not updatable.

Remarks

The FOR statement is a control statement that allows you to execute a list of SQL statements once for each row in a cursor. The FOR statement is equivalent to a compound statement with a DECLARE for the cursor and a DECLARE of a variable for each column in the result set of the cursor followed by a loop that fetches one row from the cursor into the local variables and executes *statement-list* once for each row in the cursor.

Valid cursor types include dynamic scroll (default), scroll, no scroll, sensitive, and insensitive.

The name and data type of each local variable is derived from the *statement* used in the cursor. With a SELECT statement, the data types are the data types of the expressions in the select list. The names are the select list item aliases, if they exist; otherwise, they are the names of the columns. Any select list item that is not a simple column reference must have an alias. With a CALL statement, the names and data types are taken from the RESULT clause in the procedure definition.

The LEAVE statement can be used to resume execution at the first statement after the END FOR. If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Permissions

None.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "FETCH statement [ESQL] [SP]" on page 526
- "CONTINUE statement [T-SQL]" on page 373
- "LOOP statement" on page 595

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment illustrates the use of the FOR loop.

```
FOR names AS curs INSENSITIVE CURSOR FOR
SELECT Surname
FROM Employees
DO
   CALL search_for_name( Surname );
END FOR;
```

This fragment also illustrates the use of the FOR loop.

```
BEGIN
    FOR names AS curs SCROLL CURSOR FOR
    SELECT EmployeeID, GivenName FROM Employees where EmployeeID < 130
    FOR UPDATE BY VALUES
    DO
        MESSAGE 'emp: ' || GivenName;
    END FOR;
END
```

FORWARD TO statement

Use this statement to send native syntax SQL statements to a remote server.

Syntax 1

FORWARD TO *server-name sql-statement*

Syntax 2

FORWARD TO [*server-name*]

Remarks

The FORWARD TO statement enables users to specify the server to which a passthrough connection is required. The statement can be used in two ways:

- **Syntax 1** Send a single statement to a remote server.
- **Syntax 2** Place SQL Anywhere into passthrough mode for sending a series of statements to a remote server. All subsequent statements are passed directly to the remote server. To turn passthrough mode off, issue FORWARD TO without a *server-name* specification.

  If you encounter an error from the remote server while in passthrough mode, you must still issue a FORWARD TO statement to turn passthrough off.

When establishing a connection to server-name on behalf of the user, the database server uses one of the following:

- A remote login alias set using CREATE EXTERNLOGIN
- If a remote login alias is not set up, the name and password used to communicate with SQL Anywhere

If the connection cannot be made to the server specified, the reason is contained in a message returned to the user.

After statements are passed to the requested server, any results are converted into a form that can be recognized by the client program.

server-name The name of the remote server.

SQL-statement A command in the native SQL syntax of the remote server. The command or group of commands is enclosed in curly brackets ({}) or single quotes.

> **Note**
> The FORWARD TO statement is a server directive and cannot be used in stored procedures, triggers, events, or batches.

Permissions

None

Side effects

The remote connection is set to AUTOCOMMIT (unchained) mode for the duration of the FORWARD TO session. Any work that was pending prior to the FORWARD TO statement is automatically committed.

Example

The following example sends a SQL statement to the remote server RemoteASE:

```
FORWARD TO RemoteASE { SELECT * FROM titles }
```

The following example shows a passthrough session with the remote server aseprod:

```
FORWARD TO aseprod
  SELECT * FROM titles
      SELECT * FROM authors
FORWARD TO;
```

Standards and compatibility

- **SQL/2003** Vendor extension.

FROM clause

Use this clause to specify the database tables or views involved in a SELECT, UPDATE, or DELETE statement.

Syntax

FROM *table-expression*, …

table-expression:
table-name
| *view-name*
| *procedure-name*
| *derived-table-name*
| *lateral-derived-table-name*
| *joined-table-name*
| **(** *table-expression*, … **)**

table-name :
[*userid***.**]*table-name*
[[**AS**] *correlation-name*]
[**WITH (** *table-hint* | **NO INDEX** | **INDEX (** *index-name* **))** | **FORCE INDEX (** *index-name* **)**]

view-name :
[*userid***.**] *view-name* [[**AS**] *correlation-name*]
[**WITH (** *table-hint* **)**]

procedure-name :
[*owner*.]*procedure-name* **(** [*parameter*, …] **)**
[**WITH(** *column-name data-type*, ... **)**]
[[**AS**] *correlation-name*]

derived-table-name :
(*select-statement* **)**
[**AS**] *correlation-name* [**(** *column-name*, … **)**]

lateral-derived-table-name :
LATERAL (*select-statement* | *table-expression* **)**
[**AS**] *correlation-name* [**(** *column-name*, … **)**]

joined-table-name :
table-expression join-operator table-expression
[**ON** *join-condition*]

join-operator :
[**KEY** | **NATURAL**] [*join-type*] **JOIN**
| **CROSS JOIN**

join-type:
INNER
| **LEFT** [**OUTER**]
| **RIGHT** [**OUTER**]
| **FULL** [**OUTER**]

table-hint:
HOLDLOCK
| **NOLOCK**
| **READCOMMITTED**
| **READPAST**
| **READUNCOMMITTED**
| **REPEATABLEREAD**
| **SERIALIZABLE**
| **UPDLOCK**
| **XLOCK**
| **FASTFIRSTROW**

Parameters

table-name A base table or temporary table. Tables owned by a different user can be qualified by specifying the user ID. Tables owned by groups to which the current user belongs are found by default without specifying the user ID (see "Referring to tables owned by groups" [*SQL Anywhere Server - Database Administration*]).

The WITH (INDEX (*index-name*)) clause, and the equivalent FORCE INDEX (*index-name*) clause, specify index hints for the table. It overrides the query optimizer plan selection algorithms, requiring the optimized query to access the table using the specified index, regardless of other access plans that may be available. You can specify only one index hint per correlation name. You can specify index hints only on base tables or temporary tables.

The WITH (NO INDEX) clause forces a sequential scan of the table. For example, the following SELECT statement forces the select from the Customers table to be performed sequentially:

```
SELECT * FROM Customers
   WITH ( NO INDEX )
   WHERE Customers.ID >= 500
   ORDER BY Customers.ID DESC;
```

Advanced feature
Index hints override the query optimizer, and so should be used only by experienced users. Using index hints may lead to suboptimal access plans and poor performance.

view-name Specifies a view to include in the query. As with tables, views owned by a different user can be qualified by specifying the user ID. Views owned by groups to which the current user belongs are found by default without specifying the user ID.

Although the syntax permits table hints on views, such hints have no effect.

procedure-name A stored procedure that returns a result set. Procedures can be used only in the FROM clause of SELECT statements, not UPDATE or DELETE statements. The parentheses following the procedure name are required even if the procedure does not take parameters. If the stored procedure returns multiple result sets, only the first is used.

The WITH clause provides a way of specifying column name aliases for the procedure result set. If a WITH clause is specified, the number of columns must match the number of columns in the procedure result set, and the data types must be compatible with those in the procedure result set. If no WITH clause is specified, the column names and types are those defined by the procedure definition. The following query illustrates the use of the WITH clause:


```
SELECT sp.ident, sp.quantity, Products.name
FROM ShowCustomerProducts( 149 ) WITH ( ident int, description char(20),
quantity int ) sp
   JOIN Products
ON sp.ident = Products.ID;
```

See also: "ProcCall algorithm" [*SQL Anywhere Server - SQL Usage*], and "Procedure statistics" [*SQL Anywhere Server - SQL Usage*].

derived-table-name You can supply SELECT statements instead of table or view names in the FROM clause. This allows you to use groups on groups, or joins with groups, without creating a view. The tables that you create in this way are derived tables.

lateral-derived-table-name A derived table, stored procedure, or joined table that may include outer references. You must use a lateral derived table if you want to use an outer reference in the FROM clause. For information about outer references, see "Outer references" [*SQL Anywhere Server - SQL Usage*].

You can use outer references only to tables that precede the lateral derived table in the FROM clause. For example, you cannot use an outer reference to an item in the *select-list*.

The table and the outer reference must be separated by a comma. For example, the following queries (with outer references highlighted) are valid:

```
SELECT *
 FROM A, LATERAL( B LEFT OUTER JOIN C ON ( A.x = B.x ) ) LDT;

SELECT *
 FROM A, LATERAL( SELECT * FROM B WHERE A.x = B.x ) LDT;

SELECT *
 FROM A, LATERAL( procedure-name( A.x ) ) LDT;
```

Specifying LATERAL (*table-expression*) is equivalent to specifying LATERAL (SELECT * FROM *table-expression*).

correlation-name An identifier to use when referencing an object elsewhere in the statement.

If the same correlation name is used twice for the same table in a table expression, that table is treated as if it were listed only once. For example, the following two SELECT statements are equivalent:

```
SELECT *
 FROM SalesOrders
 KEY JOIN SalesOrderItems, SalesOrders
   KEY JOIN Employees;

SELECT *
   FROM SalesOrders
   KEY JOIN SalesOrderItems
   KEY JOIN Employees;
```

Whereas the following would be treated as two instances of the Person table, with different correlation names HUSBAND and WIFE:

```
SELECT *
FROM Person HUSBAND, Person WIFE;
```


WITH table-hint The WITH *table-hint* clause allows you to specify the behavior to be used only for this table, and only for this statement. Use this clause to change the behavior without changing the isolation level or setting a database or connection option. Table hints can be used only on base tables and temporary tables.

Caution
The WITH table-hint clause is an advanced feature that should be used only if needed, and only by experienced database administrators. In addition, the setting may not be respected in all situations.

- **Isolation level related table hints** The isolation level table hints are used to specify isolation level behavior when querying tables. They specify a locking method to be used only for the specified table (s), and only for the current query. You cannot specify snapshot isolation levels as table hints.

  Following is the list of supported isolation level related table hints:

Table hint	Description
HOLDLOCK	Sets the behavior to be equivalent to isolation level 3. This table hint is synonymous with SERIALIZABLE.
NOLOCK	Sets the behavior to be equivalent to isolation level 0. This table hint is synonymous with READUNCOMMITTED.
READCOMMITTED	Sets the behavior to be equivalent to isolation level 1.
READPAST	Instructs the database server to ignore, instead of block on, rows that have write locks. Used with isolation level 1 (only). Results may vary depending on the optimization strategy used by the optimizer, particularly if the hint is specified on only a subset of the tables in the query.
READUNCOMMITTED	Sets the behavior to be equivalent to isolation level 0. This table hint is synonymous with NOLOCK.
REPEATABLEREAD	Sets the behavior to be equivalent to isolation level 2.
SERIALIZABLE	Sets the behavior to be equivalent to isolation level 3. This table hint is synonymous with HOLDLOCK.
UPDLOCK	Indicates that rows processed by the statement from the hinted table are locked using intent locks. The affected rows remain locked until the end of the transaction. UPDLOCK works at all isolation levels and uses intent locks. See "Intent locks" [*SQL Anywhere Server - SQL Usage*].

Table hint	Description
XLOCK	Indicates that rows processed by the statement from the hinted table are to be locked exclusively. The affected rows remain locked until the end of the transaction. XLOCK works at all isolation levels and uses write locks. See "Write locks" [*SQL Anywhere Server - SQL Usage*].

For information about isolation levels, see "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].

Using READPAST with MobiLink synchronization
If you are writing queries for databases that participate in MobiLink synchronization, it is recommended that you do not use the READPAST table hint in your synchronization scripts.
For more information, see:

- "download_cursor table event" [*MobiLink - Server Administration*]
- "download_delete_cursor table event" [*MobiLink - Server Administration*]
- "upload_fetch table event" [*MobiLink - Server Administration*]

If you are considering READPAST because your application performs many updates that affect download performance, an alternative solution is to use snapshot isolation. See "MobiLink isolation levels" [*MobiLink - Server Administration*].

-

Optimization table hint (FASTFIRSTROW) The FASTFIRSTROW table hint allows you to set the optimization goal for the query without setting the optimization_goal option to First-row. When you use FASTFIRSTROW, SQL Anywhere chooses an access plan that is intended to reduce the time to fetch the first row of the query's result. See "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].

Remarks

The SELECT, UPDATE, and DELETE statements require a table list to specify which tables are used by the statement.

Views and derived tables
Although the FROM clause description refers to tables, it also applies to views and derived tables unless otherwise noted.

The FROM clause creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by JOIN conditions and/or WHERE conditions.

You cannot use an ON phrase with CROSS JOIN.

Permissions

None.

Side effects

None.

See also

- "DELETE statement" on page 485
- "SELECT statement" on page 648
- "UPDATE statement" on page 703
- "Joins: Retrieving Data from Several Tables" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature, except for: KEY JOIN, which is a vendor extension; FULL OUTER JOIN and NATURAL JOIN, which are SQL/foundation features outside of core SQL; the READPAST table hint, which is a vendor extension; LATERAL (*table-expression*), which is a vendor extension (but LATERAL (*query-expression*) is in the ANSI SQL standard as feature T491)); derived tables are feature F591; procedures in the FROM clause (table functions) are feature T326; common table expressions are feature T121; recursive table expressions are feature T131. The complexity of the FROM clause means that you should check individual clauses against the standard.

Example

The following are valid FROM clauses:

```
...
FROM Employees
...

...
FROM Employees NATURAL JOIN Departments
...

...
FROM Customers
KEY JOIN SalesOrders
KEY JOIN SalesOrderItems
KEY JOIN Products
...
```

The following query illustrates how to use derived tables in a query:

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
     ( SELECT CustomerID, COUNT(*)
       FROM SalesOrders
        GROUP BY CustomerID )
     AS sales_order_counts( CustomerID,
                            number_of_orders )
ON ( Customers.ID = sales_order_counts.CustomerID )
WHERE number_of_orders > 3;
```

The following query illustrates how to select rows from stored procedure result sets:


```
SELECT t.ID, t.QuantityOrdered AS q, p.name
FROM ShowCustomerProducts( 149 ) t JOIN Products p
ON t.ID = p.ID;

SELECT *
FROM Customers WITH( readpast )
WHERE State = 'NY';
```

GET DATA statement [ESQL]

Use this statement to get string or binary data for one column of the current row of a cursor. GET DATA is usually used to fetch LONG BINARY or LONG VARCHAR fields. See "SET statement" on page 656.

Syntax

GET DATA *cursor-name*
COLUMN *column-num*
OFFSET *start-offset*
[**WITH TEXTPTR**]
USING DESCRIPTOR *sqlda-name* | **INTO** *hostvar*, …

cursor-name : *identifier*, or *hostvar*

column-num : *integer* or *hostvar*

start-offset : *integer* or *hostvar*

sqlda-name : *identifier*

Parameters

COLUMN clause The value of *column-num* starts at one, and identifies the column whose data is to be fetched. That column must be of a string or binary type.

OFFSET clause The *start-offset* indicates the number of bytes to skip over in the field value. Normally, this would be the number of bytes previously fetched. The number of bytes fetched on this GET DATA statement is determined by the length of the target host variable.

The indicator value for the target host variable is a short integer, so it cannot always contain the number of bytes truncated. Instead, it contains a negative value if the field contains the NULL value, a positive value (NOT necessarily the number of bytes truncated) if the value is truncated, and zero if a non-NULL value is not truncated.

Similarly, if a LONG VARCHAR or a LONG VARCHAR host variable is used with an offset greater than zero, the untrunc_len field does not accurately indicate the size before truncation.

WITH TEXTPTR clause If the WITH TEXTPTR clause is given, a text pointer is retrieved into a second host variable or into the second field in the SQLDA. This text pointer can be used with the Transact-SQL READ TEXT and WRITE TEXT statements. The text pointer is a 16-bit binary value, and can be declared as follows:

```
DECL_BINARY( 16 ) textptr_var;
```

WITH TEXTPTR can be used only with long data types (LONG BINARY, LONG VARCHAR, TEXT, IMAGE). If you attempt to use it with another data type, the error INVALID_TEXTPTR_VALUE is returned.

The total length of the data is returned in the SQLCOUNT field of the SQLCA structure.

Remarks

Get a piece of one column value from the row at the current cursor position. The cursor must be opened and positioned on a row, using FETCH.

Permissions

None.

Side effects

None.

See also

- "FETCH statement [ESQL] [SP]" on page 526
- "READTEXT statement [T-SQL]" on page 620

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example uses GET DATA to fetch a binary large object (also called a BLOB).

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY(1000) piece;
short ind;

EXEC SQL END DECLARE SECTION;
int size;
/* Open a cursor on a long varchar field */
EXEC SQL DECLARE big_cursor CURSOR FOR
SELECT long_data FROM some_table
WHERE key_id = 2;
EXEC SQL OPEN big_cursor;
EXEC SQL FETCH big_cursor INTO :piece;
for( offset = 0; ; offset += piece.len ) {
   EXEC SQL GET DATA big_cursor COLUMN 1
   OFFSET :offset INTO :piece:ind;
   /* Done if the NULL value */
   if( ind < 0 ) break;
   write_out_piece( piece );
   /* Done when the piece was not truncated */
   if( ind == 0 ) break;
}
EXEC SQL CLOSE big_cursor;
```


GET DESCRIPTOR statement [ESQL]

Use this statement to retrieve information about a variable within a descriptor area, or retrieves its value.

Syntax

GET DESCRIPTOR *descriptor-name*
{ *hostvar* = **COUNT** | **VALUE** { *integer* | *hostvar* } *assignment*, … }

assignment :
hostvar = **TYPE** | **LENGTH** | **PRECISION** | **SCALE** | **DATA**
| **INDICATOR** | **NAME** | **NULLABLE** | **RETURNED_LENGTH**

Remarks

The GET DESCRIPTOR statement is used to retrieve information about a variable within a descriptor area, or to retrieve its value.

The value { *integer* | *hostvar* } specifies the variable in the descriptor area about which the information is retrieved. Type checking is performed when doing GET … DATA to ensure that the host variable and the descriptor variable have the same data type. LONG VARCHAR and LONG BINARY are not supported by GET DESCRIPTOR ... DATA.

If an error occurs, it is returned in the SQLCA.

Permissions

None.

Side effects

None.

See also

- “ALLOCATE DESCRIPTOR statement [ESQL]” on page 299
- “DEALLOCATE DESCRIPTOR statement [ESQL]” on page 475
- “SET DESCRIPTOR statement [ESQL]” on page 662
- “The SQL descriptor area (SQLDA)” [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following example returns the type of the column with position col_num in sqlda.

```
int get_type( SQLDA *sqlda, int col_num )
{
    EXEC SQL BEGIN DECLARE SECTION;
    int ret_type;
    int col = col_num;
    EXEC SQL END DECLARE SECTION;
EXEC SQL GET DESCRIPTOR sqlda VALUE :col :ret_type = TYPE;
    return( ret_type );
}
```


For a longer example, see “ALLOCATE DESCRIPTOR statement [ESQL]” on page 299.

GET OPTION statement [ESQL]

You can use this statement to get the current setting of an option. It is recommended that you use the CONNECTION_PROPERTY function instead.

Syntax

GET OPTION [*userid*.]*option-name*
[**INTO** *hostvar*]
[**USING DESCRIPTOR** *sqlda-name*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*, *string*, or *hostvar*

hostvar : indicator variable allowed

sqlda-name : *identifier*

Remarks

The GET OPTION statement is provided for compatibility with older versions of the software. The recommended way to get the values of options is to use the CONNECTION_PROPERTY system function.

The GET OPTION statement gets the option setting of the option *option-name* for the user *userid* or for the connected user if *userid* is not specified. This is either the user's personal setting or the PUBLIC setting if there is no setting for the connected user. If the option specified is a database option and the user has a temporary setting for that option, then the temporary setting is retrieved.

If *option-name* does not exist, GET OPTION returns the warning SQLE_NOTFOUND.

Permissions

None required.

Side effects

None.

See also

- "SET OPTION statement" on page 664
- "System procedures" on page 835
- "CONNECTION_PROPERTY function [System]" on page 122

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement illustrates use of GET OPTION.

```
EXEC SQL GET OPTION 'date_format' INTO :datefmt;
```


GOTO statement [T-SQL]

Use this statement to branch to a labeled statement.

Syntax

label : **GOTO** *label*

Remarks

Any statement in a Transact-SQL procedure, trigger, or batch can be labeled. The label name is a valid identifier followed by a colon. In the GOTO statement, the colon is not used.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following Transact-SQL batch prints the message "yes" on the Server Messages window four times:

```
DECLARE @count SMALLINT
SELECT @count = 1
restart:
   PRINT 'yes'
   SELECT @count = @count + 1
   WHILE @count <=4
    GOTO restart
```


GRANT statement

Use this statement to create new user IDs, to grant or change permissions for the specified users, and to create or change passwords.

Syntax 1

GRANT CONNECT TO *userid*, …
[**AT** *starting-id*]
[**IDENTIFIED BY** *password*, …]

Syntax 2

GRANT *permission*, …
TO *userid*, …

permission :
DBA
| **BACKUP**
| **VALIDATE**
| **GROUP**
| **MEMBERSHIP IN GROUP** *userid*, …
| [**RESOURCE** | **ALL**]

Syntax 3

GRANT *permission*, …
ON [*owner.*]*table-name*
TO *userid*, …
[**WITH GRANT OPTION**]
[**FROM** *userid*]

permission :
ALL [**PRIVILEGES**]
| **ALTER**
| **DELETE**
| **INSERT**
| **REFERENCES** [**(** *column-name*, … **)**]
| **SELECT** [**(** *column-name*, … **)**]
| **UPDATE** [**(** *column-name*, … **)**]

Syntax 4

GRANT EXECUTE ON [*owner*.]*procedure-name*
TO *userid*, …

Syntax 5

GRANT INTEGRATED LOGIN TO *user-profile-name*, …
AS USER *userid*

Syntax 6

GRANT KERBEROS LOGIN TO *client-Kerberos-principal*, …
AS USER *userid*

Parameters

CONNECT TO Creates a new user. GRANT CONNECT can also be used by any user to change their own password. To create a user with the empty string as the password, use:

```
GRANT CONNECT TO userid IDENTIFIED BY ""
```

To create a user with no password, use:

```
GRANT CONNECT TO userid
```

A user with no password cannot connect to the database. This is useful if you are creating a group and do not want anyone to connect to the database using the group user ID. A user ID must be a valid identifier, as described in "Identifiers" on page 7. User IDs and passwords cannot:

- begin with white space, single quotes, or double quotes
- end with white space
- contain semicolons

A password can be either a valid identifier, as described in "Identifiers" on page 7, or a string (maximum 255 bytes) placed in single quotes. Passwords are case sensitive. It is recommended that the password be composed of 7-bit ASCII characters, as other characters may not work correctly if the database server cannot convert from the client's character set to UTF-8.

The verify_password_function option can be used to specify a function to implement password rules (for example, passwords must include at least one digit). If a password verification function is used, you cannot specify more than one user ID and password in the GRANT CONNECT statement. See "verify_password_function option [database]" [*SQL Anywhere Server - Database Administration*].

AT starting-id This clause is not for general purpose use. The clause specifies the internal numeric value to be used for the first user ID in the list.

The clause is implemented primarily for use by the Unload utility.

DBA Database administrator authority gives a user permission to do anything. This is usually reserved for the person in the organization who is looking after the database.

BACKUP Backup authority gives a user permission to back up the database.

VALIDATE Validate authority gives a user permission to perform the validation operations supported by the various VALIDATE statements, such as validating the database, validating tables and indexes, and validating checksums. As such, it also allows the user to use the Validation utility (dbvalid), and the Validate Database wizard in Sybase Central.

GROUP Allows the user(s) to have members. See "Managing groups" [*SQL Anywhere Server - Database Administration*].

MEMBERSHIP IN GROUP This allows the user(s) to inherit table permissions from a group and to reference tables created by the group without qualifying the table name. See "Managing groups" [*SQL Anywhere Server - Database Administration*].

Syntax 3 of the GRANT statement is used to grant permission on individual tables or views. The table permissions can be specified individually, or you can use ALL to grant all six permissions at once.

RESOURCE Allows the user to create tables and views. In syntax 2, ALL is a synonym for RESOURCE that is compatible with Sybase Adaptive Server Enterprise.

ALL In Syntax 3, this grants ALTER, DELETE, INSERT, REFERENCES, SELECT, and UPDATE permissions.

ALTER The users are allowed to alter the named table with the ALTER TABLE statement. This permission is not allowed for views.

DELETE The users are allowed to delete rows from the named table or view.

INSERT The users are allowed to insert rows into the named table or view.

REFERENCES [(column-name, ...)] The users are allowed to create indexes on the named table, and foreign keys that reference the named tables. If column names are specified, the users are allowed to reference only those columns. REFERENCES permissions on columns cannot be granted for views, only for tables. INDEX is a synonym for REFERENCES.

SELECT [(column-name, ...)] The users are allowed to look at information in the view or table. If column names are specified, the users are allowed to look at only those columns. SELECT permissions on columns cannot be granted for views, only for tables.

UPDATE [(column-name, ...)] The users are allowed to update rows in the view or table. If column names are specified, the users are allowed to update only those columns.

FROM If FROM *userid* is specified, the *userid* is recorded as a grantor user ID in the system tables. This clause is for use by the Unload utility (dbunload). Do not use or modify this option directly.

Remarks

The GRANT statement is used to grant database permissions to individual user IDs and groups. It is also used to create users and groups.

If WITH GRANT OPTION is specified, then the named user ID is also given permission to GRANT the same permissions to other user IDs. Members of groups do not inherit the WITH GRANT OPTION if it is granted to a group.

Syntax 4 of the GRANT statement is used to grant permission to execute a procedure.

Syntax 5 of the GRANT statement creates an explicit integrated login mapping between one or more Windows user or group profiles and an existing database user ID, allowing users who successfully log in to their local computer to connect to a database without having to provide a user ID or password. The *user-profile-name* can be of the form *domain\user-name*. The database user ID the integrated login is mapped to must have a password. See "Using integrated logins" [*SQL Anywhere Server - Database Administration*].

Syntax 6 of the GRANT statement creates a Kerberos authenticated login mapping from one or more Kerberos principals to an existing database user ID. This allows users who have successfully logged in to Kerberos (users who have a valid Kerberos ticket-granting ticket) to connect to a database without having to provide a user ID or password. The database user ID the Kerberos login is mapped to must have a password. The *client-Kerberos-principal* must have the format *user*/*instance*@*REALM*, where /*instance* is optional. The full principal, including the realm, must be specified, and principals that differ only in the instance or realm are treated as different.

Principals are case sensitive and must be specified in the correct case. Mappings for multiple principals that differ only in case are not supported (for example, you cannot have mappings for both jjordan@MYREALM.COM and JJordan@MYREALM.COM).

If no explicit mapping is made for a Kerberos principal, and the Guest database user ID exists and has a password, then the Kerberos principal connects using the Guest database user ID (the same Guest database user ID as for integrated logins).

For more information about Kerberos authentication, see "Using Kerberos authentication" [*SQL Anywhere Server - Database Administration*].

Permissions

Syntax 1 or 2 You must either be changing your own password using GRANT CONNECT, or have DBA authority. If you are changing another user's password (with DBA authority), the other user must not be connected to the database.

Syntax 3 If the FROM clause is specified you must have DBA authority. Otherwise, you must either own the table, or have been granted permissions on the table WITH GRANT OPTION.

Syntax 4 You must either own the procedure, or have DBA authority.

Syntax 5 or 6 You must have DBA authority.

Side effects

Automatic commit.

See also

- "REVOKE statement" on page 636

Standards and compatibility

- **SQL/2003** Syntax 3 is a core feature. Syntax 4 is a Persistent Stored Module feature. Other syntaxes are vendor extensions.

Example

Make two new users for the database.

```
GRANT
CONNECT TO Laurel, Hardy
IDENTIFIED BY Stan, Ollie;
```

Grant permissions on the Employees table to user Laurel.

```
GRANT
SELECT, UPDATE ( Street )
ON Employees
TO Laurel;
```

More than one permission can be granted in a single statement. Separate the permissions with commas.

Allow the user Hardy to execute the Calculate_Report procedure.


```
GRANT EXECUTE ON Calculate_Report
TO Hardy;
```

GRANT CONSOLIDATE statement [SQL Remote]

Use this statement to identify the database immediately above the current database in a SQL Remote hierarchy, who will receive messages from the current database.

Syntax

GRANT CONSOLIDATE
TO *userid*
TYPE *message-system*, …
ADDRESS *address-string*, …
[**SEND** { **EVERY** | **AT** } *hh*:*mm*:*ss*]

message-system:
FILE | **FTP** | **MAPI** | **SMTP** | **VIM**

address: *string*

Parameters

userid The user ID for the user to be granted the permission.

message-system One of the message systems supported by SQL Remote.

address The address for the specified message system.

Remarks

In a SQL Remote installation, the database immediately above the current database in a SQL Remote hierarchy must be granted CONSOLIDATE permissions. GRANT CONSOLIDATE is issued at a remote database to identify its consolidated database. Each database can have only one user ID with CONSOLIDATE permissions: you cannot have a database that is a remote database for more than one consolidated database.

The consolidated user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The address-name must be a valid address for the message-system, enclosed in single quotes. There can be only one consolidated user per remote database.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT CONSOLIDATE statement is required for the consolidated database to receive messages, but does not by itself subscribe the consolidated database to any data. To subscribe to data, a subscription must be created for the consolidated user ID to one of the publications in the current database. Running the database extraction utility at a consolidated database creates a remote database with the proper GRANT CONSOLIDATE statement already issued.

The optional SEND EVERY and SEND AT clauses specify a frequency at which messages are sent. The string contains a time that is a length of time between messages (for SEND EVERY) or a time of day at which messages are sent (for SEND AT). With SEND AT, messages are sent once per day.

If a user has been granted remote permissions without a SEND EVERY or SEND AT clause, the Message Agent processes messages, and then stops. To run the Message Agent continuously, you must ensure that every user with REMOTE permission has either a SEND AT or SEND EVERY frequency specified.

It is anticipated that at many remote databases, the Message Agent is run periodically, and that the consolidated database will have no SEND clause specified.

Note
Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 555
- "GRANT REMOTE statement [SQL Remote]" on page 556
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 638

Example

```
GRANT CONSOLIDATE TO con_db
TYPE mapi
ADDRESS 'Singer, Samuel';
```


GRANT PUBLISH statement [SQL Remote]

Use this statement to identify the publisher of the current database.

Syntax

GRANT PUBLISH TO *userid*

Remarks

Each database in a SQL Remote installation is identified in outgoing messages by a user ID, called the publisher. The GRANT PUBLISH statement identifies the publisher user ID associated with these outgoing messages.

Only one user ID can have PUBLISH authority. The user ID with PUBLISH authority is identified by the special constant CURRENT PUBLISHER. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER;
```

If there is no publisher, the special constant is NULL.

The current publisher special constant can be used as a default setting for columns. It is often useful to have a CURRENT PUBLISHER column as part of the primary key for replicating tables, as this helps prevent primary key conflicts due to updates at more than one site.

To change the publisher, you must first drop the current publisher using the REVOKE PUBLISH statement, and then create a new publisher using the GRANT PUBLISH statement.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “GRANT PUBLISH statement [SQL Remote]” on page 555
- “GRANT CONSOLIDATE statement [SQL Remote]” on page 553
- “REVOKE PUBLISH statement [SQL Remote]” on page 639
- “CREATE SUBSCRIPTION statement [SQL Remote]” on page 443

Example

```
GRANT PUBLISH TO publisher_ID;
```


GRANT REMOTE statement [SQL Remote]

Use this statement to identify a database immediately below the current database in a SQL Remote hierarchy, who will receive messages from the current database. These are called remote users.

Syntax

GRANT REMOTE TO *userid*, …
TYPE *message-system*, …
ADDRESS *address-string*, …
[**SEND** { **EVERY** | **AT** } *send-time*]

Parameters

userid The user ID for the user to be granted the permission

message-system One of the message systems supported by SQL Remote. It must be one of the following values:

- FILE
- FTP
- MAPI
- SMTP
- VIM

address-string A string containing a valid address for the specified message system.

send-time A string containing a time specification in the form *hh*:*mm*:*ss*.

Remarks

In a SQL Remote installation, each database receiving messages from the current database must be granted REMOTE permissions.

The single exception is the database immediately above the current database in a SQL Remote hierarchy, which must be granted CONSOLIDATE permissions.

The remote user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The address-name must be a valid address for the message-system, enclosed in single quotes.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT REMOTE statement is required for the remote database to receive messages, but does not by itself subscribe the remote user to any data. To subscribe to data, a subscription must be created for the user ID to one of the publications in the current database, using the database extraction utility or the CREATE SUBSCRIPTION statement.

The optional SEND EVERY and SEND AT clauses specify a frequency at which messages are sent. The string contains a time that is a length of time between messages (for SEND EVERY) or a time of day at which messages are sent (for SEND AT). With SEND AT, messages are sent once per day.

If a user has been granted remote permissions without a SEND EVERY or SEND AT clause, the Message Agent processes messages, and then stops. To run the Message Agent continuously, you must ensure that every user with REMOTE permission has either a SEND AT or SEND EVERY frequency specified.

It is anticipated that at many consolidated databases, the Message Agent is run continuously, so that all remote databases would have a SEND clause specified. A typical setup may involve sending messages to laptop users daily (SEND AT) and to remote servers every hour or two (SEND EVERY). You should use as few different times as possible, for efficiency.

Note
Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 555
- "REVOKE REMOTE statement [SQL Remote]" on page 640
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 553
- "Granting and revoking REMOTE and CONSOLIDATE permissions" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

- The following statement grants remote permissions to user SamS, using a MAPI email system, sending messages to the address Singer, Samuel once every two hours:

```
GRANT REMOTE TO SamS
TYPE mapi
ADDRESS 'Singer, Samuel'
SEND EVERY '02:00';
```


GRANT REMOTE DBA statement [MobiLink] [SQL Remote]

Use this statement to grant remote DBA privileges to a user ID.

Syntax

GRANT REMOTE DBA
TO *userid*, …
IDENTIFIED BY *password*

Remarks

This statement grants a limited set of DBA permissions ideal for synchronization users. The remote DBA privilege avoids having to grant full DBA privileges, thereby avoiding security problems associated with distributing DBA user IDs and passwords.

In MobiLink, REMOTE DBA authority is recommended for the SQL Anywhere synchronization client (dbmlsync). See "Permissions for dbmlsync" [*MobiLink - Client Administration*].

In SQL Remote, REMOTE DBA authority enables the Message Agent to have full access to the database to make any changes contained in the messages.

For SQL Remote, the REMOTE DBA privilege has the following properties:

- No distinct permissions when not connected from the Message Agent. A user ID granted REMOTE DBA authority has no extra privileges on any connection apart from the Message Agent. Even if the user ID and password for a REMOTE DBA user is widely distributed, there is no security problem. As long as the user ID has no permissions beyond CONNECT granted on the database, no one can use this user ID to access data in the database.
- Full DBA permissions when connected from the Message Agent.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- MobiLink: "Initiating synchronization" [*MobiLink - Client Administration*]
- SQL Remote: "The Message Agent and replication security" [*SQL Remote*]
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 641

Standards and compatibility

- **SQL/2003** Vendor extension.

GROUP BY clause

Use this clause to group columns, alias names, and functions as part of the SELECT statement.

Syntax

GROUP BY
| *group-by-term*, ...]
| *simple-group-by-term*, ... **WITH ROLLUP**
| *simple-group-by-term*, ... **WITH CUBE**
|**GROUPING SETS (** *group-by-term*, ... **)**

group-by-term :
simple-group-by-term
| **(** *simple-group-by-term*, ... **)**
| **ROLLUP (** *simple-group-by-term*, ... **)**
| **CUBE (** *simple-group-by-term*, ... **)**

simple-group-by-term :
expression
| **(** *expression* **)**
| **( )**

Parameters

GROUPING SETS clause The GROUPING SETS clause allows you to perform aggregate operations on multiple groupings from a single query specification. Each set specified in a GROUPING SET clause is equivalent to a GROUP BY clause.

For example, the following two queries are equivalent:

```
SELECT a, b, SUM( c ) FROM t
GROUP BY GROUPING SETS ( ( a, b ), ( a ), ( b ), ( ) );

SELECT a, b, SUM( c ) FROM t
  GROUP BY a, b
UNION ALL
SELECT a, NULL, SUM( c ) FROM t
  GROUP BY a
UNION ALL
SELECT NULL, b, SUM( c ) FROM t
  GROUP BY b
UNION ALL
SELECT NULL, NULL, SUM( c ) FROM t;
```

An grouping expression may be reflected in the result set as a NULL value, depending on the grouping in which the result row belongs. This may cause confusion over whether the NULL is the result of another grouping, or whether the NULL is the result of an actual NULL value in the underlying data. To distinguish between NULL values present in the input data and NULL values inserted by the grouping operator, use the GROUPING function. See "GROUPING function [Aggregate]" on page 171.

Specifying an empty set of parentheses ( ) in the GROUPING SETS clause returns a single row containing the overall aggregate.

For more information on using empty parentheses in GROUPING sets, including an example, see "Specifying an empty grouping specification" [*SQL Anywhere Server - SQL Usage*].

ROLLUP clause The ROLLUP clause is similar to the GROUPING SETS clause in that it can be used to specify multiple grouping specifications within a single query specification. A ROLLUP clause of *n* *simple-group-by-term*s generates *n*+1 grouping sets, formed by starting with the empty parentheses, and then appending successive *group-by-term*s from left to right.

For example, the following two statements are equivalent:

```
SELECT a, b, SUM( c ) FROM t
GROUP BY ROLLUP ( a, b );

SELECT a, b, SUM( c ) FROM t
GROUP BY GROUPING SETS ( ( a, b ), a, ( ) );
```

You can use a ROLLUP clause within a GROUPING SETS clause.

For more information about ROLLUP operations, see "Using ROLLUP" [*SQL Anywhere Server - SQL Usage*].

CUBE clause The CUBE clause is similar to the ROLLUP and GROUPING SETS clauses in that it can be used to specify multiple grouping specifications within a single query specification. The CUBE clause is used to represent all possible combinations that can be made from the expressions listed in the CUBE clause.

For example, the following two statements are equivalent:

```
SELECT a, b, SUM( c ) FROM t
GROUP BY CUBE ( a, b, c );

SELECT a, b, SUM( c ) FROM t
GROUP BY GROUPING SETS ( ( a, b, c ), ( a, b ), ( a, c ),
 ( b, c ), a, b, c, () );
```

You can use a CUBE clause within a GROUPING SETS clause.

For more information about ROLLUP operations, see "Using CUBE" [*SQL Anywhere Server - SQL Usage*].

WITH ROLLUP clause This is an alternative syntax to the ROLLUP clause, and is provided for T-SQL compatibility.

WITH CUBE clause This is an alternate syntax to the CUBE clause, and is provided for T-SQL compatibility.

Remarks

When using the GROUP BY clause, you can group by columns, alias names, or functions. The result of the query contains one row for each distinct value (or set of values) of each grouping set.

See also

- "SELECT statement" on page 648
- "GROUP BY clause extensions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** While the GROUP BY clause is a core feature, GROUPING SETS, ROLLUP, and CUBE are features outside of core SQL. For example, the ROLLUP clause is part of feature T431. WITH ROLLUP and WITH CUBE are vendor extensions.

Examples

The following example returns a result set showing the total number of orders, and then provides subtotals for the number of orders in each year (2000 and 2001).

```
SELECT year ( OrderDate ) Year, Quarter ( OrderDate ) Quarter, count(*)
Orders
FROM SalesOrders
GROUP BY ROLLUP ( Year, Quarter )
ORDER BY Year, Quarter;
```

Like the preceding ROLLUP operation example, the following CUBE query example returns a result set showing the total number of orders and provides subtotals for the number of orders in each year (2000 and 2001). Unlike ROLLUP, this query also gives subtotals for the number of orders in each quarter (1, 2, 3, and 4).

```
SELECT year (OrderDate) Year, Quarter ( OrderDate ) Quarter, count(*) Orders
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

The following example returns a result set that gives subtotals for the number of orders in the years 2000 and 2001. The GROUPING SETS operation lets you select the columns to be subtotaled instead of returning all combinations of subtotals like the CUBE operation.

```
SELECT year (OrderDate) Year, Quarter ( OrderDate ) Quarter, count(*) Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```


HELP statement [Interactive SQL]

Use this statement to receive help in the Interactive SQL environment.

Syntax

HELP ['*topic*']

Remarks

The HELP statement is used to access SQL Anywhere documentation.

The *topic* for help can be optionally specified. You must enclose *topic* in single quotes. In some help formats, the topic cannot be specified; in this case, a link to a general help page for Interactive SQL appears.

You can specify the following *topic* values:

- SQL Anywhere error codes
- SQL statement keywords (such as INSERT, UPDATE, SELECT, CREATE DATABASE)

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Vendor extension.

IF statement

Use this statement to control conditional execution of SQL statements.

Syntax

IF *search-condition* **THEN** *statement-list*
[**ELSEIF** { *search-condition* | *operation-type* } **THEN** *statement-list*] …
[**ELSE** *statement-list*]
END IF

Remarks

The IF statement is a control statement that allows you to conditionally execute the first list of SQL statements whose *search-condition* evaluates to TRUE. If no *search-condition* evaluates to TRUE, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed.

Execution resumes at the first statement after the END IF.

IF statement is different from IF expression
Do not confuse the syntax of the IF statement with that of the IF expression.
For information on the IF expression, see "IF expressions" on page 17.

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 351
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]
- "Search conditions" on page 20

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following procedure illustrates the use of the IF statement:

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
   SELECT CompanyName, CAST(    sum(SalesOrderItems.Quantity *
   Products.UnitPrice) AS INTEGER) VALUE
   FROM Customers
   LEFT OUTER JOIN SalesOrders
   LEFT OUTER JOIN SalesOrderItems
   LEFT OUTER JOIN Products
```



```
      GROUP BY CompanyName;
      DECLARE ThisValue INT;
      DECLARE ThisCompany CHAR(35);
      SET TopValue = 0;
      OPEN curThisCust;
      CustomerLoop:
      LOOP
         FETCH NEXT curThisCust
         INTO ThisCompany, ThisValue;
         IF SQLSTATE = err_notfound THEN
            LEAVE CustomerLoop;
         END IF;
         IF ThisValue > TopValue THEN
            SET TopValue = ThisValue;
            SET TopCompany = ThisCompany;
         END IF;
      END LOOP CustomerLoop;
      CLOSE curThisCust;
   END
```


IF statement [T-SQL]

Use this statement to control conditional execution of a SQL statement, as an alternative to the Watcom-SQL IF statement.

Syntax

IF *expression statement*
[**ELSE** [**IF** *expression*] *statement*]

Remarks

The Transact-SQL IF conditional and the ELSE conditional each control the execution of only a single SQL statement or compound statement (between the keywords BEGIN and END).

In comparison to the Watcom-SQL IF statement, there is no THEN in the Transact-SQL IF statement. The Transact-SQL version also has no ELSEIF or END IF keywords.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following example illustrates the use of the Transact-SQL IF statement:

```
IF (SELECT max(ID) FROM sysobjects) < 100
   RETURN
ELSE
         BEGIN
      PRINT 'These are the user-created objects'
      SELECT name, type, ID
      FROM sysobjects
      WHERE ID < 100
   END
```

The following two statement blocks illustrate Transact-SQL and Watcom-SQL compatibility:

```
/* Transact-SQL IF statement */
IF @v1 = 0
   PRINT '0'
ELSE IF @v1 = 1
   PRINT '1'
ELSE
   PRINT 'other'
/* Watcom-SQL IF statement */
IF v1 = 0 THEN
   PRINT '0'
ELSEIF v1 = 1 THEN
   PRINT '1'
ELSE
```



```
      PRINT 'other'
   END IF
```


INCLUDE statement [ESQL]

Use this statement to include a file into a source program to be scanned by the SQL preprocessor.

Syntax

INCLUDE *file-name*

file-name : **SQLDA** | **SQLCA** | *string*

Remarks

The INCLUDE statement is very much like the C preprocessor #include directive. The SQL preprocessor reads an embedded SQL source file and replaces all the embedded SQL statements with C-language source code. If a file contains information that the SQL preprocessor requires, include it with the embedded SQL INCLUDE statement.

Two file names are specially recognized: SQLCA and SQLDA. The following statement must appear before any embedded SQL statements in all embedded SQL source files.

```
EXEC SQL INCLUDE SQLCA;
```

This statement must appear at a position in the C program where static variable declarations are allowed. Many embedded SQL statements require variables (invisible to the programmer), which are declared by the SQL preprocessor at the position of the SQLCA include statement. The SQLDA file must be included if any SQLDAs are used.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.

INPUT statement [Interactive SQL]

Use this statement to import data into a database table from an external file or from the keyboard.

Syntax

INPUT INTO [*owner*.]*table-name*
[**FROM** *file-name* | **PROMPT**]
[**FORMAT** *input-format*]
[**ESCAPE CHARACTER** *character*]
[**ESCAPES** { **ON** | **OFF** }
[**BY ORDER** | **BY NAME**]
[**DELIMITED BY** *string*]
[**COLUMN WIDTHS (** *integer*, …**)**]
[**NOSTRIP**]
[**(** *column-name*, … **)**]
[**ENCODING** *encoding*]

input-format :
ASCII | **DBASE** | **DBASEII** | **DBASEIII**
| **EXCEL** | **FIXED** | **FOXPRO** | **LOTUS**

encoding : *identifier* or *string*

Parameters

INTO clause The name of the table into which to input the data.

FROM clause The *file-name* can be quoted or unquoted. If the string is quoted, it is subject to the same formatting requirements as other SQL strings. In particular:

- To indicate directory paths, the backslash character (\) must be represented by two backslashes. The statement to load data from the file *c:\temp\input.dat* into the Employees table is:

```
INPUT INTO Employees
FROM 'c:\\temp\\input.dat';
```

- The path name is relative to the computer Interactive SQL is running on.

PROMPT clause The PROMPT clause allows the user to enter values for each column in a row. When running in windowed mode, a dialog is displayed, allowing the user to enter the values for the new row. If you are running Interactive SQL on the command line, then Interactive SQL prompts you to type the value for each column on the command line.

FORMAT clause Each set of values must be in the format specified by the FORMAT clause, or the format set by the SET OPTION *input_format* statement if the FORMAT clause is not specified.

Certain file formats contain information about column names and types. Using this information, the INPUT statement will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: DBASEII, DBASEIII, EXCEL, FOXPRO, and LOTUS.

Input from a command file is terminated by a line containing END. Input from a file is terminated at the end of the file.

Allowable input formats are:

- **ASCII** Input lines are assumed to be characters, one row per line, with column values separated by delimiters. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing delimiters must be enclosed in either single or double quotes. If the string itself contains single or double quotes, double the quote character to use it within the string. You can use the DELIMITED BY clause to specify a different delimiter string than the default, which is a comma.

  Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single (\), and the sequence \xDD represents the character with hexadecimal code DD.

  If the file has entries indicating that a value might be null, it is treated as NULL. If the value in that position cannot be NULL, a zero is inserted in numeric columns and an empty string in character columns.
- **DBASE** The file is in dBASE II or dBASE III format. Interactive SQL will attempt to determine which format, based on information in the file. If the table does not exist, it is created.
- **DBASEII** The file is in dBASE II format. If the table does not exist, it is created.
- **DBASEIII** The file is in dBASE III format. If the table does not exist, it is created.
- **EXCEL** Input file is in the format of Microsoft Excel 2.1. If the table does not exist, it is created.
- **FIXED** Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.

  The FIXED format cannot be used with binary columns that contain embedded newline and End-of-File character sequences.
- **FOXPRO** The file is in FoxPro format. If the table does not exist, it is created.
- **LOTUS** The file is a Lotus WKS format worksheet. INPUT assumes that the first row in the Lotus WKS format worksheet is column names. If the table does not exist, it will be created. The data types used to define the new table will be selected based on the cell values in the Lotus worksheet.

ESCAPE CHARACTER clause The default escape character for hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

Newline characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X or \ is interpreted as two separate characters. For example, \q inserts a backslash and the letter q.

The escape character can be changed, using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter:

```
... ESCAPE CHARACTER '!'
```

ESCAPES clause With ESCAPES turned on (the default), characters following the escape character are interpreted as special characters by the database server. With ESCAPES turned off, the characters are read exactly as they appear in the source.

BY clause The BY clause allows the user to specify whether the columns from the input file should be matched up with the table columns based on their ordinal position in the list (ORDER, the default) or by their names (NAME). Not all input formats have column name information in the file. NAME is allowed only for those formats that do. They are the same formats that allow automatic table creation: DBASEII, DBASEIII, EXCEL, FOXPRO, and LOTUS.

DELIMITED BY clause The DELIMITED BY clause allows you to specify a string to be used as the delimiter in ASCII input format. The default delimiter is a comma.

COLUMN WIDTHS clause COLUMN WIDTHS can be specified for FIXED format only. It specifies the widths of the columns in the input file. If COLUMN WIDTHS is not specified, the widths are determined by the database column types. This clause should not be used if inserting LONG VARCHAR or BINARY data in FIXED format.

NOSTRIP clause Normally, for ASCII input format, trailing blanks are stripped from unquoted strings before the value is inserted. NOSTRIP can be used to suppress trailing blank stripping. Trailing blanks are not stripped from quoted strings, regardless of whether the option is used. Leading blanks are stripped from unquoted strings, regardless of the NOSTRIP option setting.

ENCODING clause The *encoding* argument allows you to specify the encoding that is used to read the file. The ENCODING clause can only be used with the ASCII format.

For more information on how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

For Interactive SQL, if *encoding* is not specified, the encoding that is used to read the file is determined as follows, where encoding values occurring earlier in the list take precedence over those occurring later in the list:

- the encoding specified with the default_isql_encoding option (if this option is set)
- the encoding specified with the -codepage option when Interactive SQL was started
- the default encoding for the computer Interactive SQL is running on

For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Remarks

The INPUT statement allows efficient mass insertion into a named database table. Lines of input are read either from the user via an input window (if PROMPT is specified) or from a file (if FROM *file-name* is specified). If neither is specified, the input is read from the command file that contains the INPUT statement —in Interactive SQL, this can even be directly from the SQL Statements pane.

When the input is read directly from the SQL Statements pane, you must specify a semicolon before the values for the records to be inserted at the end of the INPUT statement. For example:

```
INPUT INTO Owner.TableName;
value1, value2, value3
value1, value2, value3
value1, value2, value3
value1, value2, value3
END;
```


The END statement terminates data for INPUT statements that do not name a file and do not include the PROMPT keyword.

If a column list is specified for any input format, the data is inserted into the specified columns of the named table. By default, the INPUT statement assumes that column values in the input file appear in the same order as they appear in the database table definition. If the input file's column order is different, you must list the input file's actual column order at the end of the INPUT statement.

For example, if you create a table with the following statement:

```
CREATE TABLE inventory (
Quantity INTEGER,
item VARCHAR(60)
);
```

and you want to import ASCII data from the input file *stock.txt* that contains the name value before the quantity value,

```
'Shirts', 100
'Shorts', 60
```

then you must list the input file's actual column order at the end of the INPUT statement for the data to be inserted correctly:

```
INPUT INTO inventory
FROM stock.txt
FORMAT ASCII
(item, Quantity);
```

By default, the INPUT statement stops when it attempts to insert a row that causes an error. Errors can be treated in different ways by setting the on_error and conversion_error options (see SET OPTION). Interactive SQL prints a warning on the Messages tab if any string values are truncated on INPUT. Missing values for NOT NULL columns are set to zero for numeric types and to the empty string for non-numeric types. If INPUT attempts to insert a NULL row, the input file contains an empty row.

Because the INPUT statement is an Interactive SQL command, it cannot be used in any compound statement (such as IF) or in a stored procedure.

See “Statements allowed in procedures, triggers, events, and batches” [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have INSERT permission on the table or view.

Side effects

None.

See also

- “OUTPUT statement [Interactive SQL]” on page 604
- “INSERT statement” on page 573
- “UPDATE statement” on page 703
- “DELETE statement” on page 485
- “SET OPTION statement” on page 664

- "LOAD TABLE statement" on page 585

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following is an example of an INPUT statement from an ASCII text file.

```
INPUT INTO Employees
FROM new_emp.inp
FORMAT ASCII;
```


INSERT statement

Use this statement to insert a single row (syntax 1) or a selection of rows from elsewhere in the database (syntax 2) into a table.

Syntax 1

INSERT [**INTO**] [*owner*.]*table-name* [**(** *column-name*, … **)**]
[**ON EXISTING** { **ERROR** | **SKIP** | **UPDATE** [**DEFAULTS** { **ON | OFF** }] }]
VALUES (*expression* | **DEFAULT**, … **)**
[**OPTION(** *query-hint*, ... **)**]

Syntax 2

INSERT [**INTO**] [*owner*.]*table-name* [**(** *column-name*, … **)**]
[**ON EXISTING** { **ERROR** | **SKIP** | **UPDATE** [**DEFAULTS** { **ON | OFF** }] }]
[**WITH AUTO NAME**]
select-statement
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name = option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

WITH AUTO NAME clause WITH AUTO NAME applies only to syntax 2. If you specify WITH AUTO NAME, the names of the items in the SELECT statement determine which column the data belongs in. The SELECT statement items should be either column references or aliased expressions. Destination columns not defined in the SELECT statement are assigned their default value. This is useful when the number of columns in the destination table is very large.

ON EXISTING clause The ON EXISTING clause of the INSERT statement applies to both syntaxes. It updates existing rows in a table, based on primary key lookup, with new column values. This clause can only be used on tables that have a primary key. Attempting to use this clause on tables without primary keys generates a syntax error. You cannot insert values into a proxy table with the ON EXISTING clause.

If you specify the ON EXISTING clause, the database server performs a primary key lookup for each input row. If the corresponding row does not already exist in the table, it inserts the new row. For rows that already exist in the table, you can choose to silently ignore the input row (SKIP), generate an error message for duplicate key values (ERROR), or update the old values using the values from the input row (UPDATE). By default, if you do not specify the ON EXISTING clause, attempting to insert rows into a table where the row already exists results in a duplicate key value error, and is equivalent to specifying the ON EXISTING ERROR clause.

When using the ON EXISTING UPDATE clause, the input row is compared to the stored row. Any column values explicitly stated in the input row replace the corresponding column values in the stored row. Likewise, column values not explicitly stated in the input row result in no change to the corresponding column values in the stored row—with the exception of columns with defaults. When using the ON EXISTING UPDATE

clause with columns that have defaults (including DEFAULT AUTOINCREMENT columns), you can further specify whether to update the column value with the default values by specifying ON EXISTING UPDATE DEFAULTS ON, or leave the column value as it is by specifying ON EXISTING UPDATE DEFAULTS OFF. If nothing is specified, the default behavior is ON EXISTING UPDATE DEFAULTS OFF.

Note
DEFAULTS ON and DEFAULTS OFF parameters do not affect values in DEFAULT TIMESTAMP, DEFAULT UTC TIMESTAMP, or DEFAULT LAST USER. For these columns, the value in the stored row is always updated during the UPDATE.

When using the ON EXISTING SKIP and ON EXISTING ERROR clauses, if the table contains default columns, the server computes the default values even for rows that already exist. As a consequence, default values such as AUTOINCREMENT cause side effects even for skipped rows. In this case of AUTOINCREMENT, this results in skipped values in the AUTOINCREMENT sequence. The following example illustrates this:

```
CREATE TABLE t1( c1 INT PRIMARY KEY, c2 INT DEFAULT AUTOINCREMENT );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 20 );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 20 );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 30 );
```

The row defined in the first INSERT statement is inserted, and c2 is set to 1. The row defined in the second INSERT statement is skipped because it matches the existing row. However, the autoincrement counter still increments to 2 (but does not impact the existing row). The row defined in the third INSERT statement is inserted, and the value of c2 is set to 3. So, the values inserted for the example above are:

```
20,1
30,3
```

Caution
If you are using SQL Remote, do not replicate DEFAULT LAST USER columns. When the column is replicated the column value is set to the SQL Remote user, not the replicated value.

OPTION clause
This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query

processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].

- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The INSERT statement is used to add new rows to a database table.

Syntax 1 Insert a single row with the specified expression column values. The keyword DEFAULT can be used to cause the default value for the column to be inserted. If the optional list of column names is given, values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT *). The row is inserted into the table at an arbitrary position. (In relational databases, tables are not ordered.)

Syntax 2 Carry out mass insertion into a table with the results of a fully general SELECT statement. Insertions are done in an arbitrary order unless the SELECT statement contains an ORDER BY clause.

If you specify column names, the columns from the select list are matched ordinally with the columns specified in the column list, or sequentially in the order in which the columns were created.

Inserts can be done into views, if the query specification defining the view is updatable and has only one table in the FROM clause.

An inherently non-updatable view consists of a query expression or query specification containing any of the following:

- DISTINCT clause
- GROUP BY clause
- Aggregate function
- A *select-list* item that is not a base table.

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. Thus a string Value inserted into a table is always held in the

database with an upper-case V and the remainder of the letters lowercase. SELECT statements return the string as Value. If the database is not case sensitive, however, all comparisons make Value the same as value, VALUE, and so on. Further, if a single-column primary key already contains an entry Value, an INSERT of value is rejected, as it would make the primary key not unique.

Inserting a significant amount of data using the INSERT statement will also update column statistics.

Performance tips
To insert many rows into a table, it is more efficient to declare a cursor and insert the rows through the cursor, where possible, than to carry out many separate INSERT statements. Before inserting data, you can specify the percentage of each table page that should be left free for later updates. See "ALTER TABLE statement" on page 332.

Permissions

Must have INSERT permission on the table.

If the ON EXISTING UPDATE clause is specified, UPDATE permissions on the table are required as well.

Side effects

None.

See also

- "INPUT statement [Interactive SQL]" on page 568
- "LOAD TABLE statement" on page 585
- "UPDATE statement" on page 703
- "DELETE statement" on page 485
- "PUT statement [ESQL]" on page 614

Standards and compatibility

- **SQL/2003** Core feature. INSERT … ON EXISTING is a vendor extension.

Examples

Add an Eastern Sales department to the database.

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 230, 'Eastern Sales' );
```

Create the table DepartmentHead and fill it with the names of department heads and their departments.

```
CREATE TABLE DepartmentHead(
      pk INT PRIMARY KEY DEFAULT AUTOINCREMENT,
      DepartmentName VARCHAR(128),
      ManagerName VARCHAR(128) );
INSERT
INTO DepartmentHead (ManagerName, DepartmentName)
SELECT GivenName || ' ' || Surname, DepartmentName
FROM Employees JOIN Departments
ON EmployeeID = DepartmentHeadID;
```


Create the table DepartmentHead and fill it with the names of department heads and their departments using the WITH AUTO NAME syntax.

```
CREATE TABLE DepartmentHead(
      pk INT PRIMARY KEY DEFAULT AUTOINCREMENT,
      DepartmentName VARCHAR(128),
      ManagerName VARCHAR(128) );
INSERT
INTO DepartmentHead WITH AUTO NAME
SELECT GivenName || ' ' || Surname AS ManagerName,
      DepartmentName
FROM Employees JOIN Departments
ON EmployeeID = DepartmentHeadID;
```

Create the table MyTable and populate it using the WITH AUTO NAME syntax.

```
CREATE TABLE MyTable(
      pk INT PRIMARY KEY DEFAULT AUTOINCREMENT,
      TableName CHAR(128),
      TableNameLen INT );
INSERT into MyTable WITH AUTO NAME
SELECT
      length(t.table_name) AS TableNameLen,
      t.table_name AS TableName
FROM SYS.SYSTAB t
WHERE table_id<=10;
```

Insert a new department, executing the statement at isolation level 3, rather than using the current isolation level setting of the database.

```
INSERT INTO Departments
   (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(600, 'Foreign Sales', 129)
   OPTION( isolation_level = 3 );
```


INSTALL JAVA statement

Use this statement to make Java classes available for use within a database.

Syntax

INSTALL JAVA
[**NEW** | **UPDATE**]
[**JAR** *jar-name*]
FROM { **FILE** *file-name* | *expression* }

Parameters

NEW | UPDATE keyword If you specify an install mode of NEW, the referenced Java classes must be new classes, rather than updates of currently installed classes. An error occurs if a class with the same name exists in the database and the NEW install mode is used.

If you specify UPDATE, the referenced Java classes may include replacements for Java classes that are already installed in the given database.

If *install-mode* is omitted, the default is NEW.

JAR clause If this is specified, then the *file-name* must designate a jar file. JAR files typically have extensions of *.jar* or *.zip*.

Installed jar and zip files can be compressed or uncompressed.

If the JAR option is specified, the jar is retained as a jar after the classes that it contains have been installed. That jar is the associated jar of each of those classes. The jars installed in a database with the JAR option are called the retained jars of the database.

The *jar-name* is a character string value, of up to 255 bytes long. The *jar-name* is used to identify the retained jar in subsequent INSTALL JAVA, UPDATE, and REMOVE JAVA statements.

FROM FILE clause Specifies the location of the Java class(es) to be installed.

The formats supported for *file-name* include fully qualified file names, such as '*c:\libs\jarname.jar*' and '*/usr/u/libs/jarname.jar*', and relative file names, which are relative to the current working directory of the database server.

The *file-name* must identify either a class file, or a jar file.

FROM expression clause Expressions must evaluate to a binary type whose value contains a valid class file or jar file.

Remarks

The class definition for each class is loaded by each connection's VM the first time that class is used. When you INSTALL a class, the VM on your connection is implicitly restarted. Therefore, you have immediate access to the new class, whether the INSTALL has an *install-mode* of NEW or UPDATE. Because the VM is restarted, any values stored in Java static variables are lost, and any SQL variables with Java class types are dropped.

For other connections, the new class is loaded the next time a VM accesses the class for the first time. If the class is already loaded by a VM, that connection does not see the new class until the VM is restarted for that connection.

Permissions

DBA permissions are required to execute the INSTALL JAVA statement.

All installed classes can be referenced in any way by any user.

Not supported on Windows CE.

See also

- "REMOVE JAVA statement" on page 627

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement installs the user-created Java class named Demo, by providing the file name and location of the class.

```
INSTALL JAVA NEW
FROM FILE 'D:\JavaClass\Demo.class';
```

The following statement installs all the classes contained in a zip file, and associates them within the database with a JAR file name.

```
INSTALL JAVA
JAR 'Widgets'
FROM FILE 'C:\Jars\Widget.zip';
```

Again, the location of the zip file is not retained and classes must be referenced using the fully qualified class name (package name and class name).

INTERSECT statement

Computes the intersection between the result sets of two or more queries.

Syntax

[**WITH** *temporary-views*] *query-block*
INTERSECT [**ALL** | **DISTINCT**] *query-block*
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], …]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

Note
You cannot use the FOR, FOR XML, WITH, or OPTION clause in the *query-block*.

OPTION clause

This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].

- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The intersection between the result sets of several query blocks can be obtained as a single result using INTERSECT or INTERSECT ALL. INTERSECT DISTINCT is identical to INTERSECT.

The query blocks must each have the same number of items in the select list.

The results of INTERSECT are the same as INTERSECT ALL, except that when using INTERSECT, duplicate rows are eliminated before the intersection between the result sets is computed.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately. The first *query-block* of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None.

See also

- "EXCEPT statement" on page 513
- "UNION statement" on page 695

Standards and compatibility

- **SQL/2003** Feature F302.

Example

For examples of INTERSECT usage, see "Set operators and NULL" [*SQL Anywhere Server - SQL Usage*].

LEAVE statement

Use this statement to leave a compound statement or loop.

Syntax

LEAVE *statement-label*

Remarks

The LEAVE statement is a control statement that allows you to leave a labeled compound statement or a labeled loop. Execution resumes at the first statement after the compound statement or loop.

The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 595
- "FOR statement" on page 530
- "BEGIN statement" on page 351
- "Using Procedures, Triggers, and Batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment shows how the LEAVE statement is used to leave a loop.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters ( number )
   VALUES ( i );
   IF i >= 10 THEN
      LEAVE lbl;
   END IF;
   SET i = i + 1
END LOOP lbl
```

The following example fragment uses LEAVE in a nested loop.

```
outer_loop:
LOOP
   SET i = 1;
   inner_loop:
   LOOP
      ...
```



```
      SET i = i + 1;
      IF i >= 10 THEN
         LEAVE outer_loop
      END IF
   END LOOP inner_loop
END LOOP outer_loop
```


LOAD STATISTICS statement

For internal use only. This statement loads statistics into the ISYSCOLSTAT system table. It is used by the dbunload utility to unload column statistics from the old database. It should not be used manually.

Syntax

LOAD STATISTICS [[*owner*.]*table-name*.]*column-name*
format-id, *density*, *max-steps*, *actual-steps*, *step-values*, *frequencies*

Parameters

format_id Internal field used to determine the format of the rest of the row in the ISYSCOLSTAT system table.

density An estimate of the weighted average selectivity of a single value for the column, not counting the selectivity of large single value selectivities stored in the row.

max_steps The maximum number of steps allowed in the histogram.

actual_steps The number of steps actually used at this time.

step_values Boundary values of the histogram steps.

frequencies Selectivities of histogram steps.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ISYSCOLSTAT system table" on page 727
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

LOAD TABLE statement

Use this statement to import bulk data into a database table from an external file. *Inserts are not recorded in the log file*, raising the risk that data is lost in the event of a failure and making this statement unusable with SQL Remote or with MobiLink remote databases.

Syntax

LOAD [**INTO**] **TABLE** [*owner.*]*table-name* [**(** *column-name*, … **)**]
FROM *file-name*
[*load-option* …]
[*statistics-limitation-options*]

load-option :
CHECK CONSTRAINTS { **ON** | **OFF** }
| **COMMENTS INTRODUCED BY** *comment-prefix*
| **COMPUTES** { **ON** | **OFF** }
| **DEFAULTS** { **ON** | **OFF** }
| **DELIMITED BY** *string*
| **ENCODING** *encoding*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **ASCII** | **BCP** }
| **HEXADECIMAL** {**ON** | **OFF** }
| **ORDER** { **ON** | **OFF** }
| **PCTFREE** *percent-free-space*
| **QUOTE** *string*
| **QUOTES** { **ON** | **OFF** }
| **ROW DELIMITED BY** *string*
| **SKIP** *integer*
| **STRIP** { **ON** | **OFF** | **LTRIM** | **RTRIM** | **BOTH** }
| **WITH CHECKPOINT** { **ON** | **OFF** }

statistics-limitation-options :
STATISTICS { **ON** [**ALL COLUMNS**]
| **OFF**
| **ON KEY COLUMNS**
| **ON (** *column-list* **)** }

file-name : *string* | *variable*

comment-prefix : *string*

encoding : *string*

Parameters

Column-name Any columns not present in the column list become NULL if the DEFAULTS option is OFF. If DEFAULTS is ON and the column has a default value, that value is used. If DEFAULTS is OFF and a non-nullable column is omitted from the column list, the database server attempts to convert the empty string to the column's type.

When a column list is specified, it lists the columns that are expected to exist in the file and the order in which they are to appear. Column names cannot be repeated. Column names that do not appear in the list are set to NULL/zero/empty or DEFAULT (depending on column nullability, data type, and the DEFAULT

setting). Columns that exist in the input file that are to be ignored by LOAD TABLE can be specified using the column name "filler()".

FROM option The *file-name-string* is passed to the database server as a string. The string is therefore subject to the same database formatting requirements as other SQL strings. In particular:

- To indicate directory paths, the backslash character ('\') must be represented by two backslashes. The statement to load data from the file *c:\temp\input.dat* into the Employees table is:

  ```
  LOAD TABLE Employees
  FROM 'c:\\temp\\input.dat' ...
  ```

- The path name is relative to the database server, not to the client application. If you are running the statement on a database server on another computer, the directory names refer to directories on the database server computer, not on the client computer.
- You can use UNC path names to load data from files on computers other than the database server.

CHECK CONSTRAINTS option This option is ON by default, but the Unload utility writes out LOAD TABLE statements with the option set to OFF.

Setting CHECK CONSTRAINTS to OFF disables check constraints. This can be useful, for example, during database rebuilding. If a table has check constraints that call user-defined functions that are not yet created, the rebuild fails unless this option is set to OFF.

COMMENTS INTRODUCED BY option This option allows you to specify the string used in the data file to introduce a comment. When used, LOAD TABLE ignores any line that begins with the string *comment-prefix*.

In this example, lines in *input.dat* that start with // are ignored.

```
LOAD TABLE Employees FROM 'c:\\temp\\input.dat' COMMENTS INTRODUCED BY
'//' ...
```

Comments are only allowed at the beginning of a new line.

If the COMMENTS INTRODUCED BY option is omitted, the data file must not contain comments because they are interpreted as data.

COMPUTES option By default, COMPUTES is ON. Setting COMPUTES to ON enables recalculation of computed columns.

Setting COMPUTES to OFF disables computed column recalculations. This option is useful, for example, if you are rebuilding a database, and a table has a computed column that calls a user-defined function that is not yet created. The rebuild would fail unless this option was set to OFF.

The Unload utility (dbunload) writes out LOAD TABLE statements with the COMPUTES option set to OFF.

DEFAULTS option By default, DEFAULTS is OFF. If DEFAULTS is OFF, any column not present in the column list is assigned NULL. If DEFAULTS is OFF and a non-nullable column is omitted from the column list, the database server attempts to convert the empty string to the column's type. If DEFAULTS is ON and the column has a default value, that value is used.

DELIMITED BY option The default column delimiter string is a comma; however, it can be any string up to 255 bytes in length (for example, `... DELIMITED BY '###' ...`). The same formatting requirements apply as to other SQL strings. If you want to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `... DELIMITED BY '\x09' ...`.

ENCODING option Use the ENCODING option to specify the character encoding used for the data being loaded into the database. All data in the load file must be properly encoded in the specified character encoding. When ENCODING is not specified, the database's character encoding is used, and translation is not performed.

For more information on how to obtain the list of supported SQL Anywhere encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

If a translation error occurs during the load operation, it is reported based on the setting of the on_charset_conversion_failure option. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

The following example loads the data using UTF-8 character encoding:

```
LOAD TABLE mytable FROM 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
```

ESCAPE CHARACTER option The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter

```
... ESCAPE CHARACTER '!'
```

Only one single-byte character can be used as an escape character.

ESCAPES option With ESCAPES turned ON (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. New line characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X, or \ is interpreted as two separate characters. For example, \q inserts a backslash and the letter q.

FORMAT option If you choose ASCII, input lines are assumed to be ASCII characters, one row per line, with values separated by the column delimiter string. Choosing BCP allows the import of Adaptive Server Enterprise-generated BCP out files containing BLOBs.

HEXADECIMAL option By default, HEXADECIMAL is ON. With HEXADECIMAL ON, binary column values are read as **0x***nnnnnn*…, where each *n* is a hexadecimal digit. It is important to use HEXADECIMAL ON when dealing with multibyte character sets.

The HEXADECIMAL option can be used only with the FORMAT ASCII option.

ORDER option The default for ORDER is ON. If ORDER is ON, and a clustered index has been declared, then LOAD TABLE sorts the input data according to the clustered index and inserts rows in the same order. If the data you are loading is already sorted, you should set ORDER to OFF. See "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

PCTFREE option Specifies the percentage of free space you want to reserve for each table page. This setting overrides any permanent setting for the table, but only for the duration of the load, and only for the data being loaded.

The value percent-free-space is an integer between 0 and 100. A value of 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself.

For more information about PCTFREE, see "CREATE TABLE statement" on page 450.

QUOTE option The QUOTE clause is for ASCII data only; the *string* is placed around string values. The default is a single quote (apostrophe).

QUOTES option With QUOTES turned ON (the default), the LOAD TABLE statement expects strings to be enclosed in quote characters. The quote character is either an apostrophe (single quote) or a quotation mark (double quote). The first such character encountered in a string is treated as the quote character for the string. Strings must be terminated by a matching quote.

With QUOTES ON, column delimiter strings can be included in column values. Also, quote characters are assumed not to be part of the value. Therefore, the following line is treated as two values, not three, despite the presence of the comma in the address. Also, the quotes surrounding the address are not inserted into the database.

```
'123 High Street, Anytown',(715)398-2354
```

To include a quote character in a value, with QUOTES ON, you must use two quotes. The following line includes a value in the third column that is a single quote character:

```
'123 High Street, Anytown','(715)398-2354',''''
```

ROW DELIMITED BY option Use this clause to specify the string that indicates the end of an input record. The default delimiter string is a newline (\n); however, it can be any string up to 255 bytes in length (for example, `... ROW DELIMITED BY '###' ...`). The same formatting requirements apply as to other SQL strings. If you wanted to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `... ROW DELIMITED BY '\x09' ...`. If your delimiter string contains a \n, it will match either \r\n or \n.

SKIP option Include a SKIP option to ignore the first few lines of a file. The *integer* argument specifies the number of lines to skip. You can use this option to skip over a line containing column headings, for example. If the row delimiter is not the default (newline), then skipping may not work correctly if the data contains the row delimiter embedded within a quoted string.

STRIP option Use the STRIP option to specify whether unquoted values should have leading or trailing blanks stripped off before they are inserted. The STRIP option accepts the following options:

- **STRIP OFF** No stripping of leading or trailing blanks.
- **STRIP LTRIM** Strip leading blanks.
- **STRIP RTRIM** Strip trailing blanks.
- **STRIP BOTH** Strip both leading and trailing blanks

- **STRIP ON** Deprecated. Equivalent to STRIP RTRIM.

WITH CHECKPOINT option The default setting is OFF. If set to ON, a checkpoint is issued after successfully completing and logging the statement.

If WITH CHECKPOINT ON is not specified, and the database requires automatic recovery before a CHECKPOINT is issued, the data file used to load the table must be present for the recovery to complete successfully. If WITH CHECKPOINT ON is specified, and recovery is subsequently required, recovery begins after the checkpoint, and the data file need not be present.

Caution
If you set the database option conversion_error to Off, you may load bad data into your table without any error being reported. If you do not specify WITH CHECKPOINT ON, and the database needs to be recovered, the recovery may fail as conversion_error is On (the default value) during recovery. It is recommended that you do not load tables with conversion_error set to Off and WITH CHECKPOINT ON not specified.

See "conversion_error option [compatibility]" [*SQL Anywhere Server - Database Administration*].

The data files are required, regardless of this option, if the database becomes corrupt and you need to use a backup and apply the current log file.

statistics-limitation-options Allows you to limit the columns for which statistics are generated during the execution of LOAD TABLE. Otherwise, statistics are generated for all columns. You should only use this option if you are certain that statistics will not be used on some columns. You can specify ON ALL COLUMNS (the default), OFF, ON KEY COLUMNS, or a list of columns for which statistics should be generated.

Remarks

Caution
LOAD TABLE is intended solely for fast loading of large amounts of data. LOAD TABLE does not write individual rows to the transaction log.

LOAD TABLE allows efficient mass insertion into a database table from a file. LOAD TABLE is more efficient than the Interactive SQL statement INPUT.

LOAD TABLE places a write lock on the whole table. For base tables and global temporary tables a commit is performed. For local temporary tables, a commit is not performed

When loading data from a UTF-16 or UTF-8 data file, LOAD TABLE ignores the byte order mark (BOM) if it is present. The database server assumes that the data has the same byte order as that of the computer on which the database server is running.

Do not use the LOAD TABLE statement on a global temporary table for which ON COMMIT DELETE ROWS was specified, either explicitly or by default, at creation time. However, you *can* use LOAD TABLE if ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL was specified.

With FORMAT ASCII, a NULL value is indicated by specifying no value. For example, if three values are expected and the file contains `1,,'Fred',`, then the values inserted are 1, NULL, and 'Fred'. If the file

contains `1,2,`, then the values 1, 2, and NULL are inserted. Values that consist only of spaces are also considered NULL values. For example, if the file contains `1, ,'Fred',`, then values 1, NULL, and 'Fred' are inserted. All other values are considered not NULL. For example, '' (single-quote single-quote) is an empty string. 'NULL' is a string containing four letters.

If a column being loaded by LOAD TABLE does not allow NULL values and the file value is NULL, then numeric columns are given the value 0 (zero), character columns are given an empty string (''). If a column being loaded by LOAD TABLE allows NULL values and the file value is NULL, then the column value is NULL (for all types).

If the LOAD TABLE statement contains a column list, a column not specified in the column list is loaded as follows:

- if DEFAULT ON is specified, and the column has a default value, the default value is used.
- if the column does not have a DEFAULT value, and NULLs are allowed, a NULL is used.
- if the column has no DEFAULT value and does not allow NULLs, either a zero (0) or an empty string (''), is used, or an error is returned, depending on the data type of the column.

LOAD TABLE and column statistics In order to create histograms on table columns, LOAD TABLE captures column statistics when it loads data. The histograms are used by the optimizer For more information on how column statistics are used by the optimizer, see "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*].

Following are additional tips about loading and column statistics:

- LOAD TABLE saves statistics on base tables for future use. It does not save statistics on global temporary tables.
- If you are loading into an empty table that may have previously contained data, it may be beneficial to drop statistics for the column before executing the LOAD TABLE statement. See "DROP STATISTICS statement" on page 508.
- It is important to note that if column statistics exist when a LOAD TABLE is performed on a column, statistics for the column are not recalculated. Instead, statistics for the new data is inserted into the existing statistics. This means that if the existing column statistics are out-of-date, they will still be out of date after loading new data into the column. If you suspect that the column statistics are out of date, you should consider updating them either before, or after, executing the LOAD TABLE statement. See "Updating column statistics" [*SQL Anywhere Server - SQL Usage*].
- LOAD TABLE adds statistics only if the table has five or more rows. If the table has at least five rows, histograms are modified as follows:

Data already in table?	Histogram present?	Action taken
Yes	Yes	Integrate changes into the existing histograms
Yes	No	Do not build histograms

Data already in table?	Histogram present?	Action taken
No	Yes	Integrate changes into the existing histograms
No	No	Build new histograms

- LOAD TABLE does not generate statistics for columns that contain NULL values for more than 90% of the rows being loaded.

Using dynamically constructed file names You can execute a LOAD TABLE statement with a dynamically constructed file name by assigning the file name to a variable and using the variable name in the LOAD TABLE statement.

Permissions

The permissions required to execute a LOAD TABLE statement depend on the database server –gl option, as follows:

- If the -gl option is set to ALL, you must be the owner of the table or have DBA authority or have ALTER privileges.
- If the -gl option is set to DBA, you must have DBA authority.
- If the -gl option is set to NONE, LOAD TABLE is not permitted.

See "-gl server option" [*SQL Anywhere Server - Database Administration*].

Requires an exclusive lock on the table.

Side effects

Automatic commit except for local temporary tables

Inserts are not recorded in the log file. Thus, the inserted rows may not be recovered in the event of a failure. In addition, the LOAD TABLE statement should never be used in a database involved in SQL Remote replication or databases used as MobiLink clients because these technologies replicate changes through analysis of the log file.

The LOAD TABLE statement does not fire any triggers associated with the table.

A checkpoint is carried out at the beginning of the operation. A second checkpoint, at the end of the operation, is optional.

Column statistics are updated if a significant amount of data is loaded.

See also

- "UNLOAD TABLE statement" on page 700
- "INSERT statement" on page 573
- "INPUT statement [Interactive SQL]" on page 568

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Following is an example of LOAD TABLE. First, you create a table, and then load data into it using a file called *input.txt*.

```
CREATE TABLE t( a CHAR(100), let_me_default INT DEFAULT 1, c CHAR(100) );
```

Following is the content of a file called *input.txt*:

```
ignore_me, this_is_for_column_c, this_is_for_column_a
```

The following LOAD statement loads the file called *input.txt*:

```
LOAD TABLE T ( filler(), c, a ) FROM 'input.txt' FORMAT ASCII DEFAULTS ON;
```

The command `SELECT * FROM t` yields the result set:

```
this_is_for_column_a, 1, this_is_for_column_c
```

Execute the LOAD TABLE statement with a dynamically-constructed file name, via the EXECUTE IMMEDIATE statement:

```
CREATE PROCEDURE LoadData( IN from_file LONG VARCHAR )
BEGIN
    DECLARE path LONG VARCHAR;
    SET path = 'd:\\data\\' || from_file;
    LOAD MyTable FROM path;
END;
```

The following example loads UTF-8-encoded table data into mytable:

```
LOAD TABLE mytable FROM 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
```

LOCK TABLE statement

Use this statement to prevent other concurrent transactions from accessing or modifying a table.

Syntax

LOCK TABLE *table-name*
[**WITH HOLD**]
IN { **SHARE** | **EXCLUSIVE** } **MODE**

Parameters

table-name The table must be a base table, not a view. As temporary table data is local to the current connection, locking global or local temporary tables has no effect.

WITH HOLD clause If this clause is specified, the lock is held until the end of the connection. If the clause is not specified, the lock is released when the current transaction is committed or rolled back.

SHARE mode Prevent other transactions from modifying the table, but allow them read access. Transactions can change data while in SHARE mode, provided no other transaction holds a lock of any kind on the row(s) being modified.

EXCLUSIVE mode Prevent other transactions from accessing the table. No other transaction can execute queries, updates of any kind, or any other action against the table. If a table t is locked exclusively with a statement such as `LOCK TABLE t IN EXCLUSIVE MODE`, the default server behavior is to not acquire row locks for *t*. This behavior can be disabled by setting the subsume_row_locks option to Off.

Remarks

The LOCK TABLE statement allows direct control over concurrency at a table level, independent of the current isolation level.

While the isolation level of a transaction generally governs the kinds of locks that are set when the current transaction executes a request, the LOCK TABLE statement allows more explicit control locking of the rows in a table.

Permissions

To lock a table in SHARE mode, SELECT privileges are required.

To lock a table in EXCLUSIVE mode; you must be the table owner or have DBA authority.

Side effects

Other transactions that require access to the locked table may be delayed or blocked.

See also

- "SELECT statement" on page 648
- "sa_locks system procedure" on page 882

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement prevents other transactions from modifying the Customers table for the duration of the current transaction:

```
LOCK TABLE Customers
IN SHARE MODE;
```

LOOP statement

Use this statement to repeat the execution of a statement list.

Syntax

[*statement-label* :]
[**WHILE** *search-condition*] **LOOP**
statement-list
END LOOP [*statement-label*]

Remarks

The WHILE and LOOP statements are control statements that allow you to execute a list of SQL statements repeatedly while a *search-condition* evaluates to TRUE. The LEAVE statement can be used to resume execution at the first statement after the END LOOP.

If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Permissions

None.

Side effects

None.

See also

- "FOR statement" on page 530
- "CONTINUE statement [T-SQL]" on page 373

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

A While loop in a procedure.

```
...
SET i = 1;
WHILE i <= 10 LOOP
   INSERT INTO Counters( number ) VALUES ( i );
   SET i = i + 1;
END LOOP;
...
```

A labeled loop in a procedure.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters( number )
   VALUES ( i );
   IF i >= 10 THEN
      LEAVE lbl;
   END IF;
```



```
      SET i = i + 1;
   END LOOP lbl
```


MESSAGE statement

Use this statement to display a message.

Syntax

MESSAGE *expression*, …
[**TYPE** { **INFO** | **ACTION** | **WARNING** | **STATUS** }]
[**TO** { **CONSOLE**
  | **CLIENT** [**FOR** { **CONNECTION** *conn_id* | **ALL** }]
  | [**EVENT** | **SYSTEM**] **LOG** }
  [**DEBUG ONLY**]
]

conn_id : integer

Parameters

TYPE clause This clause specifies the message type. Acceptable values are INFO, ACTION, WARNING, and STATUS. The client application must decide how to handle the message. For example, Interactive SQL displays messages in the following locations:

- **INFO** The Messages tab. INFO is the default type.
- **ACTION** A Message box with an OK button.
- **WARNING** A Message box with an OK button.
- **STATUS** The Messages tab.

TO clause This clause specifies the destination of a message:

- **CONSOLE** Send messages to the Server Messages window. CONSOLE is the default.
- **CLIENT** Send messages to the client application. Your application must decide how to handle the message, and you can use the TYPE as information on which to base that decision.
- **LOG** Send messages to the server log file specified by the -o option. If EVENT or SYSTEM is specified, the message is also written to the console and to the Windows event log under event source SQLANY 10.0 Admin and to the Unix SysLog under the name SQLANY 10.0 Admin (servername). Messages in the server log are identified as follows:
  - **i** Messages of type INFO or STATUS.
  - **w** Messages of type WARNING.
  - **e** Messages of type ACTION.

FOR clause For messages TO CLIENT, this clause specifies which connections receive notification about the message:

- **CONNECTION conn_id** Specify the recipient's connection ID for the message.
- **ALL** Specify that all open connections receive the message.

DEBUG ONLY This clause allows you to control whether debugging messages added to stored procedures and triggers are enabled or disabled by changing the setting of the debug_messages option. When DEBUG ONLY is specified, the MESSAGE statement is executed only when the debug_messages option is set to On.

> **Note**
> DEBUG ONLY messages are inexpensive when the debug_messages option is set to Off, so these statements can usually be left in stored procedures on a production system. However, they should be used sparingly in locations where they would be executed frequently; otherwise, they may result in a small performance penalty.

Remarks

The MESSAGE statement displays a message, which can be any expression. Clauses can specify the message type and where the message appears.

The procedure issuing a MESSAGE … TO CLIENT statement must be associated with a connection.

For example, the message box is not displayed in the following example because the event occurs outside of a connection.

```
CREATE EVENT CheckIdleTime
TYPE ServerIdle
WHERE event_condition( 'IdleTime' ) > 100
HANDLER
BEGIN
   MESSAGE 'Idle engine' type warning to client;
END;
```

However, in the following example, the message is written to the server console.

```
CREATE EVENT CheckIdleTime
TYPE ServerIdle
WHERE event_condition( 'IdleTime' ) > 100
HANDLER
BEGIN
   MESSAGE 'Idle engine' type warning to console;
END;
```

Valid expressions can include a quoted string or other constant, variable, or function.

The FOR clause can be used to notify another application of an event detected on the database server without the need for the application to explicitly check for the event. When the FOR clause is used, recipients receive the message the next time that they execute a SQL statement. If the recipient is currently executing a SQL statement, the message is received when the statement completes. If the statement being executed is a stored procedure call, the message is received before the call is completed.

If an application requires notification within a short time after the message is sent and when the connection is not executing SQL statements, you can use a second connection. This connection can execute one or more WAITFOR DELAY statements. These statements do not consume significant resources on the server or network (as would happen with a polling approach), but permit applications to receive notification of the message shortly after it is sent.

Embedded SQL and ODBC clients receive messages via message callback functions. In each case, these functions must be registered. In embedded SQL, the message callback is registered with db_register_a_callback using the DB_CALLBACK_MESSAGE parameter. In ODBC, the message callback is registered with SQLSetConnectAttr using the ASA_REGISTER_MESSAGE_CALLBACK parameter.

Permissions

DBA authority is required to execute a MESSAGE statement containing a FOR clause or a TO EVENT LOG or TO SYSTEM LOG clause.

Side effects

None.

See also

- "CREATE PROCEDURE statement" on page 414
- "debug_messages option [database]" [*SQL Anywhere Server - Database Administration*]
- "db_register_a_callback function" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

1. The following procedure displays a message on the Server Messages window:

```
CREATE PROCEDURE message_text()
BEGIN
MESSAGE 'The current date and time: ', Now();
END;
```

   The statement following statement displays the string `The current date and time`, followed by the current date and time, on the database Server Messages window.

```
CALL message_text();
```

2. To register a callback in ODBC, declare the message handler:

```
void SQL_CALLBACK my_msgproc(
   VOID * sqlca,
   UNSIGNED CHAR msg_type,
   LONG code,
   UNSIGNED SHORT len,
   CHAR* msg )
{ … }
```

   Note that msg is not null-terminated. Your application must be designed to hand this.

3. Install the declared message handler by calling the SQLSetConnectAttr function:

```
rc = SQLSetConnectAttr(
   dbc,
   SA_REGISTER_MESSAGE_CALLBACK,
   (SQLPOINTER) &my_msgproc, SQL_IS_POINTER );
```

1. To register a callback in embedded SQL, first define the message handler:


```
void SQL_CALLBACK my_msgproc(
   SQLCA * sqlca,
   UNSIGNED CHAR msg_type,
   LONG code,
   UNSIGNED SHORT len,
   CHAR* msg ) // msg is NOT null terminated
{ ... }
```

2. Install the declared message handler by calling the db_register_a_callback function:

```
db_register_a_callback( &sqlca, DB_CALLBACK_MESSAGE, (SQL_CALLBACK_PARM)
&my_msgproc );
```


OPEN statement [ESQL] [SP]

Use this statement to open a previously declared cursor to access information from the database.

Syntax

OPEN *cursor-name*
[**USING** [**DESCRIPTOR** *sqlda-name* | *hostvar*, ...]]
[**WITH HOLD**]
[**ISOLATION LEVEL** *n*]
[**BLOCK** *n*]

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

Parameters

Embedded SQL usage After successful execution of the OPEN statement, the *sqlerrd*[*3*] field of the SQLCA (SQLIOESTIMATE) is filled in with an estimate of the number of input/output operations required to fetch all rows of the query. Also, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) is filled with either the actual number of rows in the cursor (a value greater than or equal to 0), or an estimate thereof (a negative number whose absolute value is the estimate). It is the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows (see "row_counts option [database]" [*SQL Anywhere Server - Database Administration*]), but this can be expensive.

If *cursor-name* is specified by an identifier or string, the corresponding DECLARE CURSOR must appear prior to the OPEN in the C program; if the *cursor-name* is specified by a host variable, the DECLARE CURSOR statement must execute before the OPEN statement.

USING DESCRIPTOR clause The USING DESCRIPTOR clause is for embedded SQL only. It specifies the host variables to be bound to the place-holder bind variables in the SELECT statement for which the cursor has been declared.

WITH HOLD clause By default, all cursors are automatically closed at the end of the current transaction (COMMIT or ROLLBACK). The optional WITH HOLD clause keeps the cursor open for subsequent transactions. It will remain open until the end of the current connection or until an explicit CLOSE statement is executed. Cursors are automatically closed when a connection is terminated.

ISOLATION LEVEL clause The ISOLATION LEVEL clause allows this cursor to be opened at an isolation level different from the current setting of the isolation_level option. All operations on this cursor are performed at the specified isolation level regardless of the option setting. If this clause is not specified, then the cursor's isolation level for the entire time the cursor is open is the value of the isolation_level option when the cursor is opened. See "How locking works" [*SQL Anywhere Server - SQL Usage*].

The following values are supported:

- 0
- 1
- 2
- 3

- snapshot
- statement snapshot
- readonly statement snapshot

The cursor is positioned before the first row (see "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*] or "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]).

BLOCK clause This clause is for embedded SQL use only. Rows may be fetched by the client application more than one at a time. This is referred to as block fetching, prefetching, or multi-row fetching. The BLOCK clause can reduce the number of rows prefetched. Specifying the BLOCK clause on OPEN is the same as specifying the BLOCK clause on each FETCH. See "FETCH statement [ESQL] [SP]" on page 526.

Remarks

The OPEN statement opens the named cursor. The cursor must be previously declared.

When the cursor is on a CALL statement, OPEN causes the procedure to execute until the first result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set.

Permissions

Must have SELECT permission on all tables in a SELECT statement, or EXECUTE permission on the procedure in a CALL statement.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "RESUME statement" on page 633
- "PREPARE statement [ESQL]" on page 610
- "FETCH statement [ESQL] [SP]" on page 526
- "RESUME statement" on page 633
- "CLOSE statement [ESQL] [SP]" on page 363
- "FOR statement" on page 530

Standards and compatibility

- **SQL/2003** Embedded SQL use is a core feature. Procedures use is a Persistent Stored Modules feature.

Example

The following examples show the use of OPEN in embedded SQL.

```
EXEC SQL OPEN employee_cursor;
```

and

```
EXEC SQL PREPARE emp_stat FROM
'SELECT empnum, empname FROM Employees WHERE name like ?';
EXEC SQL DECLARE employee_cursor CURSOR FOR emp_stat;
EXEC SQL OPEN employee_cursor USING :pattern;
```

The following example is from a procedure or trigger.


```
BEGIN
   DECLARE cur_employee CURSOR FOR
   SELECT Surname
   FROM Employees;
   DECLARE name CHAR(40);
   OPEN cur_employee;
   LP: LOOP
         FETCH NEXT cur_employee INTO name;
         IF SQLCODE <> 0 THEN LEAVE LP END IF;
    ...
   END LOOP
   CLOSE cur_employee;
END
```


OUTPUT statement [Interactive SQL]

Use this statement to output the current query results to a file.

Syntax

OUTPUT TO *file-name*
[**APPEND**]
[**VERBOSE**]
[**FORMAT** *output-format*]
[**ESCAPE CHARACTER** *character*]
[**ESCAPES** { **ON** | **OFF** }
[**DELIMITED BY** *string*]
[**QUOTE** *string* [**ALL**]]
[**COLUMN WIDTHS (***integer*, …**)**]
[**HEXADECIMAL** { **ON** | **OFF** | **ASIS** }]
[**ENCODING** *encoding*]

output-format :
ASCII | **DBASEII** | **DBASEIII** | **EXCEL**
| **FIXED** | **FOXPRO** | **HTML** | **LOTUS**
| **SQL** | **XML**

encoding : *string* or *identifier*

Parameters

APPEND clause This optional keyword is used to append the results of the query to the end of an existing output file without overwriting the previous contents of the file. If the APPEND clause is not used, the OUTPUT statement overwrites the contents of the output file by default. The APPEND keyword is valid if the output format is ASCII, FIXED, or SQL.

VERBOSE clause When the optional VERBOSE keyword is included, error messages about the query, the SQL statement used to select the data, and the data itself are written to the output file. Lines that do not contain data are prefixed by two hyphens. If VERBOSE is omitted (the default) only the data is written to the file. The VERBOSE keyword is valid if the output format is ASCII, FIXED, or SQL.

FORMAT clause Allowable output formats are:

- **ASCII** The output is an ASCII format file with one row per line in the file. All values are separated by commas, and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the DELIMITED BY and QUOTE clauses. If ALL is specified in the QUOTE clause, all values (not just strings) are quoted.

  Three other special sequences are also used. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD. This is the default output format.

- **DBASEII** The output is a dBASE II format file which includes column definitions. Note that a maximum of 32 columns can be output. Column names are truncated to 11 characters, and each row of data in each column is truncated to 255 characters.

- **DBASEIII** The output is a dBASE III format file which includes column definitions. Note that a maximum of 128 columns can be output. Column names are truncated to 11 characters, and each row of data in each column is truncated to 255 characters.
- **EXCEL** The output is an Excel 2.1 worksheet. The first row of the worksheet contains column labels (or names if there are no labels defined). Subsequent worksheet rows contain the actual table data.
- **FIXED** The output is fixed format with each column having a fixed width. The width for each column can be specified using the COLUMN WIDTHS clause. No column headings are output in this format.

  If the COLUMN WIDTHS clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. The exception is that LONG VARCHAR and LONG BINARY data default to 32 KB.
- **FOXPRO** The output is a FoxPro format file which includes column definitions. Note that a maximum of 128 columns can be output. Column names are truncated to 11 characters. Column names are truncated to 11 characters, and each row of data in each column is truncated to 255 characters.
- **HTML** The output is in the Hyper Text Markup Language format.
- **LOTUS** The output is a Lotus WKS format worksheet. Column names are put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file Interactive SQL can produce.
- **SQL** The output is an Interactive SQL INPUT statement required to recreate the information in the table.
- **XML** The output is an XML file encoded in UTF-8 and containing an embedded DTD. Binary values are encoded in CDATA blocks with the binary data rendered as 2-hex-digit strings. The INPUT statement does not accept XML as a file format.

ESCAPE CHARACTER clause The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter

```
... ESCAPE CHARACTER '!'
```

New line characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X, or \ is interpreted as two separate characters. For example, \q inserts a backslash and the letter q.

ESCAPES clause With ESCAPES turned on (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. With ESCAPES turned off, the characters are written exactly as they appear in the source.

DELIMITED BY clause The DELIMITED BY clause is for the ASCII output format only. The delimiter string is placed between columns (default comma).

QUOTE clause The QUOTE clause is for the ASCII output format only. The quote string is placed around string values. The default is a single quote character. If ALL is specified in the QUOTE clause, the quote string is placed around all values, not just around strings.

COLUMN WIDTHS clause The COLUMN WIDTHS clause is used to specify the column widths for the FIXED format output.

HEXADECIMAL clause The HEXADECIMAL clause specifies how binary data is to be unloaded for the ASCII format only. When set to ON, binary data is unloaded in the format 0xabcd. When set to OFF, binary data is escaped when unloaded (\xab\xcd). When set to ASIS, values are written as is, that is, without any escaping—even if the value contains control characters. ASIS is useful for text that contains formatting characters such as tabs or carriage returns.

ENCODING clause The *encoding* argument allows you to specify the encoding that is used to write the file. The ENCODING clause can only be used with the ASCII format.

For more information on how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

With Interactive SQL, if *encoding* is not specified, the encoding that is used to write the file is determined as follows, where encoding values occurring earlier in the list take precedence over those occurring later in the list:

- the encoding specified with the default_isql_encoding option (if this option is set)
- the encoding specified with the -codepage option when Interactive SQL was started
- the default encoding for the computer Interactive SQL is running on

For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Remarks

The OUTPUT statement copies the information retrieved by the current query to a file.

The output format can be specified with the optional FORMAT clause. If no FORMAT clause is specified, the Interactive SQL output_format option setting is used (see "output_format option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*]).

The current query is the statement that generated the information that appears on the Results tab in the Results pane. The OUTPUT statement reports an error if there is no current query.

Because the INPUT statement is an Interactive SQL command, it cannot be used in any compound statement (such as IF) or in a stored procedure. See "Statements allowed in procedures, triggers, events, and batches" [*SQL Anywhere Server - SQL Usage*].

Permissions

None.

Side effects

In Interactive SQL, the Results tab displays only the results of the current query. All previous query results are replaced with the current query results.

See also

- "SELECT statement" on page 648
- "INPUT statement [Interactive SQL]" on page 568
- "UNLOAD TABLE statement" on page 700

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

Place the contents of the Employees table in a file in ASCII format:

```
SELECT *
FROM Employees;
OUTPUT TO Employees.txt
FORMAT ASCII;
```

Place the contents of the Employees table at the end of an existing file, and include any messages about the query in this file as well:

```
SELECT *
FROM Employees;
OUTPUT TO Employees.txt APPEND VERBOSE;
```

Suppose you need to export a value that contains an embedded line feed character. A line feed character has the numeric value 10, which you can represent as the string '\x0a' in a SQL statement. If you execute the following statement, with HEXADECIMAL set to ON,

```
SELECT 'line1\x0aline2';
OUTPUT TO file.txt HEXADECIMAL ON;
```

you get a file with one line in it containing the following text:

```
line10x0aline2
```

But if you execute the same statement with HEXADEMICAL set to OFF, you get the following:

```
line1\x0aline2
```

Finally, if you set HEXADECIMAL to ASIS, you get a file with two lines:

```
line1
line2
```

You get two lines when you use ASIS because the embedded line feed character has been exported without being converted to a two digit hex representation, and without being prefixed by anything.

PARAMETERS statement [Interactive SQL]

Use this statement to specify parameters to an Interactive SQL command file.

Syntax

PARAMETERS *parameter1*, *parameter2*, …

Remarks

The PARAMETERS statement names the parameters for a command file, so that they can be referenced later in the command file.

Parameters are referenced by putting `{parameter1}` into the file where you want the named parameter to be substituted. There must be no spaces between the braces and the parameter name.

If a command file is invoked with less than the required number of parameters, Interactive SQL prompts for values of the missing parameters.

Permissions

None.

Side effects

None.

See also

- "READ statement [Interactive SQL]" on page 618

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following Interactive SQL command file takes two parameters.

```
PARAMETERS department_id, file;
SELECT Surname
FROM Employees
WHERE DepartmentID = { department_id }
>#{file}.dat;
```

If you save this script in a file named *test.sql*, you can run it from Interactive SQL using the following command:

```
READ test.sql [100] [data]
```


PASSTHROUGH statement [SQL Remote]

Use this statement to start or stop passthrough mode for SQL Remote administration. Forms 1 and 2 start passthrough mode, while form 3 stops passthrough mode.

Syntax 1

PASSTHROUGH [**ONLY**] **FOR** *userid*, …

Syntax 2

PASSTHROUGH [**ONLY**] **FOR SUBSCRIPTION**
TO [*owner.*]*publication-name* [**(** *constant* **)**]

Syntax 3

PASSTHROUGH STOP

Remarks

In passthrough mode, any SQL statements are executed by the database server, and are also placed into the transaction log to be sent in messages to subscribers. If the ONLY keyword is used to start passthrough mode, the statements are not executed at the server; they are sent to recipients only. When a passthrough session contains calls to stored procedures, the procedures must exist in the server that is issuing the passthrough commands, even if they are not being executed locally at the server. The recipients of the passthrough SQL statements are either a list of user IDs (syntax 1) or all subscribers to a given publication. Passthrough mode may be used to apply changes to a remote database from the consolidated database or send statements from a remote database to the consolidated database.

Syntax 2 sends statements to remote databases whose subscriptions are started, and does not send statements to remote databases whose subscriptions are created and not started.

Permissions

Must have DBA authority.

Side effects

None.

Example

```
PASSTHROUGH FOR rem_db ;
...
( SQL statements to be executed at the remote database )
...
PASSTHROUGH STOP ;
```


PREPARE statement [ESQL]

Use this statement to prepare a statement to be executed later, or used to define a cursor.

Syntax

PREPARE *statement-name*
  FROM *statement*
[**DESCRIBE** *describe-type* **INTO** [[**SQL**] **DESCRIPTOR**] *descriptor*]
[**WITH EXECUTE**]

statement-name : *identifier* or *hostvar*

statement : *string* or *hostvar*

describe-type :
 [**ALL** | **BIND VARIABLES** | **INPUT** | **OUTPUT** | **SELECT LIST**]
 [**LONG NAMES** [[[**OWNER.**]**TABLE.**]**COLUMN**]
     | **WITH VARIABLE RESULT**]

Parameters

statement-name The statement name can be an identifier or host variable. However, you should not use an identifier when using multiple SQLCAs. If you do, two prepared statements may have the same statement number, which could cause the wrong statement to be executed or opened. Also, using an identifier for a statement name is not recommended for multi-threaded applications where the statement name may be referenced by multiple threads concurrently.

DESCRIBE clause If DESCRIBE INTO DESCRIPTOR is used, the prepared statement is described into the specified descriptor. The describe type may be any of the describe types allowed in the DESCRIBE statement.

WITH EXECUTE clause If the WITH EXECUTE clause is used, the statement is executed if and only if it is not a CALL or SELECT statement, and it has no host variables. The statement is immediately dropped after a successful execution. If the PREPARE and the DESCRIBE (if any) are successful but the statement cannot be executed, a warning SQLCODE 111, SQLSTATE 01W08 is set, and the statement is not dropped.

The DESCRIBE INTO DESCRIPTOR and WITH EXECUTE clauses may improve performance because they cut down on the required client/server communication.

WITH VARIABLE RESULT clause The WITH VARIABLE RESULT clause is used to describe procedures that may have more than one result set, with different numbers or types of columns.

If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the describe to one of the following values:

- **0** The result set may change: The procedure call should be described again following each OPEN statement.
- **1** The result set is fixed. No redescribing is required.

Static and dynamic
For compatibility reasons, preparing COMMIT, PREPARE TO COMMIT, and ROLLBACK statements is still supported. However, it is recommended that you do all transaction management operations with static embedded SQL because certain application environments may require it. Also, other embedded SQL systems do not support dynamic transaction management operations.

Remarks

The PREPARE statement prepares a SQL statement from the *statement* and associates the prepared statement with *statement-name*. This statement name is referenced to execute the statement, or to open a cursor if the statement is a SELECT statement. The *statement-name* may be a host variable of type a_sql_statement_number defined in the *sqlca.h* header file that is automatically included. If an identifier is used for the *statement-name*, only one statement per module may be prepared with this *statement-name*.

If a host variable is used for *statement-name*, it must have the type SHORT INT. There is a typedef for this type in *sqlca.h* called a_sql_statement_number. This type is recognized by the SQL preprocessor and can be used in a DECLARE section. The host variable is filled in by the database during the PREPARE statement, and need not be initialized by the programmer.

Permissions

None.

Side effects

Any statement previously prepared with the same name is lost.

The statement is dropped after use only if you use WITH EXECUTE and the execution is successful. You should ensure that you DROP the statement after use in other circumstances. If you do not, the memory associated with the statement is not reclaimed.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "DESCRIBE statement [ESQL]" on page 490
- "OPEN statement [ESQL] [SP]" on page 601
- "EXECUTE statement [ESQL]" on page 515
- "DROP statement" on page 498

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement prepares a simple query:

```
EXEC SQL PREPARE employee_statement FROM
'SELECT Surname FROM Employees';
```


PREPARE TO COMMIT statement

Use this statement to check whether a COMMIT can be performed successfully.

Syntax

PREPARE TO COMMIT

Remarks

The PREPARE TO COMMIT statement tests whether a COMMIT can be performed successfully. The statement will cause an error if a COMMIT is impossible without violating the integrity of the database.

The PREPARE TO COMMIT statement cannot be used in stored procedures, triggers, events, or batches.

Permissions

None.

Side effects

None.

See also

- "COMMIT statement" on page 367
- "ROLLBACK statement" on page 642

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following sequence of statements leads to an error because of foreign key checking on the Employees table.

```
EXECUTE IMMEDIATE
   "SET OPTION wait_for_commit = 'On'";
EXECUTE IMMEDIATE "DELETE FROM Employees
   WHERE EmployeeID = 160";
EXECUTE IMMEDIATE "PREPARE TO COMMIT";
```

The following sequence of statements does not cause an error when the delete statement is executed, even though it causes integrity violations. The PREPARE TO COMMIT statement returns an error.

```
SET OPTION wait_for_commit= 'On';
DELETE
FROM Departments
WHERE DepartmentID = 100;
PREPARE TO COMMIT;
```

PRINT statement [T-SQL]

Use this statement to return a message to the client, or display a message in the message window of the database server.

Syntax

PRINT *format-string* [, *arg-list*]

Remarks

The PRINT statement returns a message to the client window if you are connected from an Open Client application or jConnect application. If you are connected from an embedded SQL or ODBC application, the message is displayed on the Server Messages window.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Permissions

None.

Side effects

None.

See also

- "MESSAGE statement" on page 597

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement displays a message:

```
PRINT 'Display this message';
```

The following statement illustrates the use of placeholders in the PRINT statement:

```
DECLARE @var1 INT, @var2 INT
SELECT @var1 = 3, @var2 = 5
PRINT 'Variable 1 = %1!, Variable 2 = %2!', @var1, @var2
```


PUT statement [ESQL]

Use this statement to insert a row into the specified cursor.

Syntax

PUT *cursor-name*
[**USING DESCRIPTOR** *sqlda-name* | **FROM** *hostvar-list*]
[**INTO** { **DESCRIPTOR** *sqlda-name* | *hostvar-list* }]
[**ARRAY** :*nnn*]

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

hostvar-list : may contain indicator variables

Remarks

Inserts a row into the named cursor. Values for the columns are taken from the first SQLDA or the host variable list, in a one-to-one correspondence with the columns in the INSERT statement (for an INSERT cursor) or the columns in the select list (for a SELECT cursor).

The PUT statement can be used only on a cursor over an INSERT or SELECT statement that references a single table in the FROM clause, or that references an updatable view consisting of a single base table.

If the sqldata pointer in the SQLDA is the null pointer, no value is specified for that column. If the column has a DEFAULT VALUE associated with it, that is used; otherwise, a NULL value is used.

The second SQLDA or host variable list contains the results of the PUT statement.

The optional ARRAY clause can be used to carry out wide puts, which insert more than one row at a time and which may improve performance. The value nnn is the number of rows to be inserted. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

Inserting into a cursor
For scroll (values sensitive) cursors, the inserted row will appear if the new row matches the WHERE clause and the keyset cursor has not finished populating. For dynamic cursors, if the inserted row matches the WHERE clause, the row may appear. Insensitive cursors cannot be updated.

For information on putting LONG VARCHAR or LONG BINARY values into the database, see "SET statement" on page 656.

Permissions

Must have INSERT permission.

Side effects

When inserting rows into a value-sensitive (keyset driven) cursor, the inserted rows appear at the end of the result set, even when they do not match the WHERE clause of the query or if an ORDER BY clause would

normally have placed them at another location in the result set. See "Modifying rows through a cursor" [*SQL Anywhere Server - Programming*].

See also

- "UPDATE statement" on page 703
- "UPDATE (positioned) statement [ESQL] [SP]" on page 708
- "DELETE statement" on page 485
- "DELETE (positioned) statement [ESQL] [SP]" on page 488
- "INSERT statement" on page 573

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement illustrates the use of PUT in embedded SQL:

```
EXEC SQL PUT cur_employee FROM :employeeID, :surname;
```


RAISERROR statement [T-SQL]

Use this statement to signal an error and to send a message to the client.

Syntax

RAISERROR *error-number* [*format-string*] [, *arg-list*]

Parameters

error-number The *error-number* is a five-digit integer greater than 17000. The error number is stored in the global variable @@error.

format-string If *format-string* is not supplied or is empty, the error number is used to locate an error message in the system tables. Adaptive Server Enterprise obtains messages 17000-19999 from the SYSMESSAGES table. In SQL Anywhere this table is an empty view, so errors in this range should provide a format string. Messages for error numbers of 20000 or greater are obtained from the ISYSUSERMESSAGE table.

In SQL Anywhere, the *format-string* length can be up to 255 bytes.

The extended values supported by the Adaptive Server Enterprise RAISERROR statement are not supported in SQL Anywhere.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form %*nn*!, where *nn* is an integer between 1 and 20.

Intermediate RAISERROR status and code information is lost after the procedure terminates. If at return time an error occurs along with the RAISERROR then the error information is returned and the RAISERROR information is lost. The application can query intermediate RAISERROR statuses by examining @@error global variable at different execution points.

Remarks

The RAISERROR statement allows user-defined errors to be signaled and sends a message on the client.

Permissions

None.

Side effects

None.

See also

- "CREATE TRIGGER statement [T-SQL]" on page 468
- "on_tsql_error option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "continue_after_raiserror option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement raises error 23000, which is in the range for user-defined errors, and sends a message to the client. Note that there is no comma between the *error-number* and the *format-string* parameters. The first item following a comma is interpreted as the first item in the argument list.

```
RAISERROR 23000 'Invalid entry for this column: %1!', @val
```

The next example uses RAISERROR to disallow connections.

```
CREATE PROCEDURE DBA.login_check()
BEGIN
    // Allow a maximum of 3 concurrent connections
    IF( DB_PROPERTY('ConnCount') > 3 ) THEN
         RAISERROR 28000
      'User %1! is not allowed to connect -- there are ' ||
                        'already %2! users logged on',
      Current User,
      CAST( DB_PROPERTY( 'ConnCount' ) AS INT )-1;
    ELSE
         CALL sp_login_environment;
    END IF;
END
go
GRANT EXECUTE ON DBA.login_check TO PUBLIC
go
SET OPTION PUBLIC.login_procedure='DBA.login_check'
go
```

For an alternate way to disallow connections, see “login_procedure option [database]” [*SQL Anywhere Server - Database Administration*].

READ statement [Interactive SQL]

Use this statement to read Interactive SQL statements from a file.

Syntax

READ [**ENCODING** *encoding*] *file-name* **[** *parameter* **]** ...

encoding : *identifier* or *string*

Remarks

The READ statement reads a sequence of Interactive SQL statements from the named file. This file can contain any valid Interactive SQL statements, including other READ statements. READ statements can be nested to any depth. If the file name does not contain an absolute path, Interactive SQL searches for the file. Interactive SQL will first search the current directory, and then the directories specified in the environment variable SQLPATH, and then the directories specified in the environment variable PATH. If the named file has no file extension, Interactive SQL searches each directory for the same file name with the extension *.sql*.

The *encoding* argument allows you to specify the encoding that is used to read the file. The READ statement does not process escape characters when it reads a file. It assumes that the entire file is in the specified encoding.

If *encoding* is not specified, Interactive SQL determines the encoding that is used to read the file as follows, where encoding values occurring earlier in the list take precedence over those occurring later in the list:

- the encoding specified with the default_isql_encoding option (if this option is set)
- the encoding specified with the -codepage option when Interactive SQL was started
- the default encoding for the computer Interactive SQL is running on

For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Parameters can be listed after the name of the command file. These parameters correspond to the parameters named on the PARAMETERS statement at the beginning of the statement file (see "PARAMETERS statement [Interactive SQL]" on page 608). Parameter names must be enclosed in square brackets. Interactive SQL substitutes the corresponding parameter wherever the source file contains {*parameter-name*}, where*parameter-name* is the name of the appropriate parameter.

The parameters passed to a command file can be identifiers, numbers, quoted identifiers, or strings. When quotes are used around a parameter, the quotes are put into the text during the substitution. Parameters that are not identifiers, numbers, or strings (contain spaces or tabs) must be enclosed in square brackets ([ ]). This allows for arbitrary textual substitution in the command file.

If not enough parameters are passed to the command file, Interactive SQL prompts for values for the missing parameters.

Permissions

None.

Side effects

None.

See also

- "PARAMETERS statement [Interactive SQL]" on page 608

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following are examples of the READ statement.

```
READ status.rpt '160'
READ birthday.sql [>= '1988-1-1'] [<= '1988-1-30']
```


READTEXT statement [T-SQL]

Use this statement to read text and image values from the database, starting from a specified offset and reading a specified number of bytes.

Syntax

READTEXT *table-name.column-name*
text-pointer offset size
[**HOLDLOCK**]

Remarks

READTEXT is used to read image and text values from the database. You cannot perform READTEXT operations on views.

Permissions

SELECT permissions on the table.

Side effects

None.

See also

- "WRITETEXT statement [T-SQL]" on page 722
- "GET DATA statement [ESQL]" on page 542
- "TEXTPTR function [Text and image]" on page 265

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

REFRESH MATERIALIZED VIEW statement

Initializes or refreshes the data in a materialized view by executing its query definition.

Syntax

REFRESH MATERIALIZED VIEW [*owner*.]*materialized-view-name*
[**WITH** { **ISOLATION LEVEL** *isolation-level* | **EXCLUSIVE MODE** }]
[**FORCE BUILD**]

isolation-level :
0 | **1** | **2** | **3** | **snapshot** | **statement-snapshot** | **readonly-statement-snapshot**

Parameters

WITH ISOLATION LEVEL isolation-level clause Use this clause to change the isolation level for the execution of the refresh operation. For information on isolation levels, see "Using Transactions and Isolation Levels" [*SQL Anywhere Server - SQL Usage*], and "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].

WITH EXCLUSIVE MODE clause Use this clause if you do not want to change the isolation level, but do want to guarantee that the data is updated to be consistent with committed data in the underlying tables. When using WITH EXCLUSIVE MODE, table locks are placed on all tables referenced by the materialized view to prevent data in those tables from changing while the materialized view is refreshed. However, other connections can still read data from the underlying tables. If table locks cannot be obtained, the refresh operation fails and an error is returned.

FORCE BUILD clause By default, when you execute the REFRESH MATERIALIZED VIEW statement, the database server checks whether the underlying data has changed. If it hasn't, the materialized view is not refreshed. Specify FORCE BUILD to refresh the materialized view, regardless of whether underlying data has changed.

Remarks

Use this statement to initialize a materialized view, or to refresh data in a materialized view. Refreshing means that the database re-executes the query definition for the view, and replaces the materialized view data with the new data that is returned, thereby making the materialized view data consistent with the data in the underlying tables. By default, the database server refreshes the materialized view using the current isolation level set for the connection.

Several options need to have specific values in order to refresh a materialized view, and in order for the view to be used in optimization. Additionally, there are options that are remembered for each materialized view; these options must match the current options in order to refresh the view, or to use the view in optimization. For more information about the options that must have specific settings, see "Restrictions when managing materialized views" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have INSERT permission on the materialized view, and SELECT permission on the tables in the materialized view definition.

Not supported within snapshot transactions. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Any open cursors that reference the materialized view are closed.

A checkpoint is performed at the beginning of execution.

Automatic commits are performed at the beginning and end of execution.

While executing, an exclusive lock is placed on the materialized view being refreshed using the connection BLOCKING option, and shared table locks, without blocking, are placed on all tables referenced by the materialized view. Also, until refreshing is complete, the materialized view is in an uninitialized state, making it unavailable to the database server or optimizer.

See also

- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "CREATE MATERIALIZED VIEW statement" on page 411
- "ALTER MATERIALIZED VIEW statement" on page 313

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement changes the isolation level of the connection to 1 (read committed), and then refreshes the data in the ProductIDsPerCustomer materialized view by forcing the view to be rebuilt:

```
REFRESH MATERIALIZED VIEW ProductIDsPerCustomer
WITH ISOLATION LEVEL 1
FORCE BUILD;
```

The original isolation level is restored at the end of the statement execution.

REFRESH TRACING LEVEL statement

Use the REFRESH TRACING LEVEL statement to reload the tracing levels from the sa_diagnostic_tracing_level table while a tracing session is in progress.

Syntax

REFRESH TRACING LEVEL

Remarks

This statement is used to reload the tracing level information from the sa_diagnostic_tracing_level table. It must be called from the database being profiled.

When a tracing session is first started, rows from the sa_diagnostic_tracing_level table are loaded into server memory to control what kind of information is traced. If you want to change the types of data being traced, without stopping and restarting the tracing session to do so, you can do so by manually deleting or inserting the appropriate rows in the sa_diagnostic_tracing_level table, and then executing the REFRESH TRACING LEVEL statement to reload the settings.

To see the current tracing levels, query the sa_diagnostic_tracing_level table as follows:

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

For more information on the sa_diagnostic_tracing_level system table, see "sa_diagnostic_tracing_level table" on page 748.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ATTACH TRACING statement" on page 344
- "DETACH TRACING statement" on page 496
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you are troubleshooting a performance problem. You turn on a high level of tracing for the entire database to capture the queries that are causing the problem. After starting the tracing session, you find that capturing all queries for all users on your system slows down your database too much, so you decide you'd rather limit tracing to one user and wait for that user to report a problem. However, you do not want to stop the tracing session to change the settings.

You can do this in Sybase Central by using the Database Tracing wizard, which is the recommended method. However, you can also do this from the command line by replacing the rows in sa_diagnostic_tracing_level

table where scope=DATABASE and enabled=1, with equivalent rows where scope=USER, identifier=*userid*, enabled=1, and so on. Then, you execute a REFRESH TRACING LEVEL statement to continue tracing using use the new settings.

RELEASE SAVEPOINT statement

Use this statement to release a savepoint within the current transaction.

Syntax

RELEASE SAVEPOINT [*savepoint-name*]

Remarks

Release a savepoint. The *savepoint-name* is an identifier specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is released.

Releasing a savepoint does not do any type of COMMIT. It simply removes the savepoint from the list of currently active savepoints.

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

- "BEGIN TRANSACTION statement [T-SQL]" on page 354
- "COMMIT statement" on page 367
- "ROLLBACK statement" on page 642
- "ROLLBACK TO SAVEPOINT statement" on page 643
- "SAVEPOINT statement" on page 647
- "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

REMOTE RESET statement [SQL Remote]

Use this statement in custom database-extraction procedures to start all subscriptions for a remote user in a single transaction.

Syntax

REMOTE RESET *userid*

Remarks

This command starts all subscriptions for a remote user in a single transaction. It sets the log_sent and confirm_sent values in ISYSREMOTEUSER table to the current position in the transaction log. It also sets the created and started values in ISYSSUBSCRIPTION to the current position in the transaction log for all subscriptions for this remote user. The statement does not do a commit. You must do an explicit commit after this call.

To write an extraction process that is safe on a live database, the data must be extracted at isolation level 3 in the same transaction as the subscriptions are started.

This statement is an alternative to start subscription. START SUBSCRIPTION has an implicit commit as a side effect, so that if a remote user has several subscriptions, it is impossible to start them all in one transaction using START SUBSCRIPTION.

Permissions

Must have DBA authority.

Side effects

No automatic commit is done by this statement.

See also

- "START SUBSCRIPTION statement [SQL Remote]" on page 679
- "ISYSREMOTEUSER system table" on page 732

Example

- The following statement resets the subscriptions for remote user SamS:

```
REMOTE RESET SamS;
```


REMOVE JAVA statement

Use this statement to remove a class or a jar file from a database. When a class is removed it is no longer available for use as a column or variable type.

The class or jar must already be installed.

Syntax

REMOVE JAVA *classes-to-remove*

classes-to-remove :
CLASS *java-class-name*, … | **JAR** *jar-name*, …

Parameters

CLASS The *java-class-name* parameter is the name of one or more Java class to be removed. These classes must be installed classes in the current database.

JAR The *jar-name* is a character string value of maximum length 255.

Each *jar-name* must be equal to the *jar-name* of a retained jar in the current database. Equality of *jar-name* is determined by the character string comparison rules of the SQL system.

Remarks

Removes a class or jar file from the database.

Permissions

Must have DBA authority.

Not supported on Windows CE.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement removes a Java class named Demo from the current database.

```
REMOVE JAVA CLASS Demo;
```


REORGANIZE TABLE statement

Use this statement to defragment tables when a full rebuild of the database is not possible due to the requirements for continuous access to the database.

Syntax

REORGANIZE TABLE [*owner*.]*table-name*
[{ **PRIMARY KEY**
| **FOREIGN KEY** *foreign-key-name*
| **INDEX** *index-name* }
| **ORDER** {**ON** | **OFF**}
]

Parameters

PRIMARY KEY Reorganizes the primary key index for the table.

FOREIGN KEY Reorganizes the specified foreign key.

INDEX Reorganizes the specified index.

ORDER option With ORDER ON (the default), the data is ordered by clustered index if one exists. If a clustered index does not exist, the data is ordered by primary key values. With ORDER OFF, the data is ordered by primary key.

For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

Remarks

Table fragmentation can impede performance. Use this statement to defragment rows in a table, or to compress indexes which have become sparse due to DELETEs. It may also reduce the total number of pages used to store the table and its indexes, and it may reduce the number of levels in an index tree. However, it will not result in a reduction of the total size of the database file. It is recommended that you use the sa_table_fragmentation and sa_index_density system procedures to select tables worth processing.

If an index or key is not specified, the reorganization process defragments rows in the table by deleting and re-inserting groups of rows. For each group, an exclusive lock on the table is obtained. Once the group has been processed, the lock is released and re-acquired (waiting if necessary), providing an opportunity for other connections to access the table. Checkpoints are suspended while a group is being processed; once a group is finished, a checkpoint may occur. The rows are processed in order by primary key (if it exists) or clustered index; if the table has no primary key or clustered index, an error results. The processed rows are re-inserted at the end of the table, resulting in the rows being clustered by primary key at the end of the process. Note that the same amount of work is required, regardless of how fragmented the rows initially were.

If an index or key is specified, the specified index is processed. For the duration of the operation, an exclusive lock is held on the table and checkpoints are suspended. Any attempts to access the table by other connections will block or fail, depending on their setting of the blocking option. The duration of the lock is minimized by pre-reading the index pages prior to obtaining the exclusive lock.

Since both forms of reorganization may modify many pages, the checkpoint log can become large. This can result in a increase in the database file size. However, this increase is temporary since the checkpoint log is deleted at shutdown and the file is truncated at that point.

This statement is not logged to the transaction log.

Permissions

- Must be either the owner of the table, or a user with DBA authority.
- Not supported on Windows CE.
- Not supported within snapshot transactions. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Prior to starting the reorganization, a checkpoint is done to try to maximize the number of free pages. Also, when executing the REORGANIZE TABLE statement, there is an implied commit for approximately every 100 rows, so reorganizing a large table causes multiple commits to take place.

Examples

The following statement reorganizes the primary key index for the Employees table:

```
REORGANIZE TABLE Employees
PRIMARY KEY;
```

The following statement reorganizes the table pages of the Employees table:

```
REORGANIZE TABLE  Employees;
```

The following statement reorganizes the index IX_product_name on the Products table:

```
REORGANIZE TABLE Products
   INDEX IX_product_name;
```

The following statement reorganizes the foreign key FK_DepartmentID_DepartmentID for the Employees table:

```
REORGANIZE TABLE Employees
   FOREIGN KEY FK_DepartmentID_DepartmentID;
```


RESIGNAL statement

Use this statement to resignal an exception condition.

Syntax

RESIGNAL [*exception-name*]

Remarks

Within an exception handler, RESIGNAL allows you to quit the compound statement with the exception still active, or to quit reporting another named exception. The exception is handled by another exception handler or returned to the application. Any actions by the exception handler before the RESIGNAL are undone.

Permissions

None.

Side effects

None.

See also

- "SIGNAL statement" on page 673
- "BEGIN statement" on page 351
- "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]
- "RAISERROR statement [T-SQL]" on page 616

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment returns all exceptions except Column Not Found to the application.

```
...
DECLARE COLUMN_NOT_FOUND EXCEPTION
   FOR SQLSTATE '52003';
...
EXCEPTION
WHEN COLUMN_NOT_FOUND THEN
SET message='Column not found';
WHEN OTHERS THEN
RESIGNAL;
```


RESTORE DATABASE statement

Use this statement to restore a backed up database from an archive.

Syntax

RESTORE DATABASE *file-name*
FROM *archive-root*
[**CATALOG ONLY**
| [**RENAME** *dbspace-name* **TO** *new-dbspace-name*] …]
[**HISTORY** { **ON** | **OFF** }]

file-name : *string* | *variable*
archive-root : *string* | *variable*
new-dbspace-name : *string* | *variable*

Parameters

CATALOG ONLY clause Retrieve information about the named archive, and place it in the backup history file (*backup.syb*), but do not restore any data from the archive.

RENAME clause Specifies a new location to restore each dbspace to.

HISTORY clause By default, each RESTORE DATABASE operation appends a line to the *backup.syb*. You can prevent updates to the *backup.syb* file by specifying HISTORY OFF. You may want to prevent the file from being updated if all of the following conditions apply:

- your RESTORE DATABASE operations occur frequently
- there is no procedure to periodically archive or delete the *backup.syb* file
- disk space is very limited

Remarks

Each RESTORE DATABASE operation updates a history file called *backup.syb*.

For more information about the *backup.syb* file, see "SALOGDIR environment variable" [*SQL Anywhere Server - Database Administration*].

The RENAME clause provides a way to change the restore location for each dbspace. The dbspace name in a RENAME clause cannot be SYSTEM or TRANSLOG. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

RESTORE DATABASE replaces the database that is being restored. If you need incremental backups, use the image format of the BACKUP command and save only the transaction log; however, image backups to tape are not supported.

Permissions

The permissions required to execute this statement are set on the server command line, using the -gu option. The default setting is to require DBA authority. See "-gu server option" [*SQL Anywhere Server - Database Administration*].

This statement is not supported on Windows CE.

Side effects

None.

See also

- "BACKUP statement" on page 346
- "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.
- **Windows CE** Not supported on Windows CE.

Example

The following example restores a database from a tape drive. The number of backslashes that are required depends on which database you are connected to when you execute RESTORE DATABASE. The database affects the setting of the escape_character option. It is normally set to On, but is set to Off in utility_db. When connected to any database other than utility_db, the extra backslashes are required.

```
RESTORE DATABASE 'd:\\dbhome\\mydatabase.db'
FROM '\\\\.\\tape0';
```


RESUME statement

Use this statement to resume execution of a cursor that returns result sets.

Syntax

RESUME *cursor-name*

cursor-name : *identifier* | *hostvar*

Remarks

This statement resumes execution of a procedure that returns result sets. The procedure executes until the next result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set. This warning is also set when you RESUME a cursor for a SELECT statement.

The RESUME statement is not supported in Interactive SQL. If you want to view multiple result sets in Interactive SQL, you can set the isql_show_multiple_result_sets option to ON, or choose Tools ▶ Options, and then select Show Multiple Result sets on the Results tab.

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 478
- "FETCH statement [ESQL] [SP]" on page 526
- "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Following are embedded SQL examples.

```
1. EXEC SQL RESUME cur_employee;
2. EXEC SQL RESUME :cursor_var;
```

RETURN statement

Use this statement to exit from a function, procedure or batch unconditionally, optionally providing a return value.

Syntax

RETURN [*expression*]

Remarks

A RETURN statement causes an immediate exit from a block of SQL. If *expression* is supplied, the value of *expression* is returned as the value of the function or procedure.

If the RETURN appears inside an inner BEGIN block, it is the outer BEGIN block that is terminated.

Statements following a RETURN statement are not executed.

Within a function, the expression should be of the same data type as the function's RETURNS data type.

Within a procedure, RETURN is used for Transact-SQL-compatibility, and is used to return an integer error code.

Permissions

None.

Side effects

None.

See also

- "CREATE FUNCTION statement" on page 399
- "CREATE PROCEDURE statement" on page 414
- "BEGIN statement" on page 351

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following function returns the product of three numbers:

```
CREATE FUNCTION product (
   a NUMERIC,
   b NUMERIC,
   c NUMERIC )
RETURNS NUMERIC
BEGIN
   RETURN ( a * b * c );
END;
```

Calculate the product of three numbers:

```
SELECT product(2, 3, 4);
```


product(2, 3, 4)
24

The following procedure uses the RETURN statement to avoid executing a complex query if it is meaningless:

```
CREATE PROCEDURE customer_products
( in customer_ID integer DEFAULT NULL)
RESULT ( ID integer, quantity_ordered integer )
BEGIN
   IF customer_ID NOT IN (SELECT ID FROM Customers)
   OR customer_ID IS NULL THEN
      RETURN
   ELSE
      SELECT Products.ID,sum(
         SalesOrderItems.Quantity )
      FROM  Products,
            SalesOrderItems,
            SalesOrders
      WHERE SalesOrders.CustomerID=customer_ID
      AND SalesOrders.ID=SalesOrderItems.ID
      AND SalesOrderItems.ProductID=Products.ID
      GROUP BY Products.ID
   END IF
END;
```

REVOKE statement

Use this statement to remove permissions from users.

Syntax 1

REVOKE *permission*, … **FROM** *userid*, …

permission :
CONNECT
| **DBA**
| **BACKUP**
| **VALIDATE**
| **INTEGRATED LOGIN**
| **KERBEROS LOGIN**
| **GROUP**
| **MEMBERSHIP IN GROUP** *userid*, …
| **RESOURCE**

Syntax 2

REVOKE *table-permission*, …
ON [*owner*.]*table-name*
FROM *userid*, …

table-permission :
ALL [**PRIVILEGES**]
| **ALTER**
| **DELETE**
| **INSERT**
| **REFERENCES** [**(** *column-name*, … **)**]
| **SELECT** [**(** *column-name*, … **)**]
| **UPDATE** [**(** *column-name*, … **)**]

Syntax 3

REVOKE EXECUTE
ON [*owner*.]*procedure-name*
FROM *userid*, …

Remarks

The REVOKE statement removes permissions given using the GRANT statement. Syntax 1 revokes special user permissions. Syntax 2 revokes table permissions. Syntax 3 revokes permission to execute a procedure.

REVOKE CONNECT removes a user ID from a database, and also destroys any objects (tables, views, procedures, and so on) owned by that user and any permissions granted by that user. You cannot execute a REVOKE CONNECT on a user if the user being dropped owns a table referenced by a view owned by another user.

REVOKE GROUP automatically revokes MEMBERSHIP IN GROUP from all members of the group.

When you add a user to a group, the user inherits all the permissions assigned to that group. SQL Anywhere does not allow you to revoke a subset of the permissions that a user inherits as a member of a group because you can only revoke permissions that are explicitly given by a GRANT statement. If you need to have

different permissions for different users, you can create different groups with the appropriate permissions, or you can explicitly grant each user the permissions they require.

When you grant or revoke group permissions for tables, views, or procedures, all members of the group inherit those changes. The DBA, RESOURCE, and GROUP permissions are not inherited: you must assign them to each individual user ID that requires them.

If you give a user WITH GRANT OPTION permission, and later revoke that permission, you also revoke any permissions that user granted to others while they had the WITH GRANT OPTION permission.

Permissions

Must be the grantor of the permissions that are being revoked or have DBA authority.

If you are revoking connect permissions or table permissions from another user, the other user must not be connected to the database. You cannot revoke connect permissions from DBO.

When you are connected to the utility database, executing REVOKE CONNECT FROM DBA disables future connections to the utility database. This means that no future connections can be made to the utility database unless you use a connection that existed before the REVOKE CONNECT was done, or restart the database server.

Side effects

Automatic commit.

See also

- "GRANT statement" on page 548

Standards and compatibility

- **SQL/2003** Syntax 1 is a vendor extension. Syntax 2 is a core feature. Syntax 3 is a Persistent Stored Modules feature.

Example

Prevent user Dave from updating the Employees table.

```
REVOKE UPDATE ON Employees FROM Dave;
```

Revoke resource permissions from user Jim.

```
REVOKE RESOURCE FROM Jim;
```

Revoke an integrated login mapping from the user profile named Administrator.

```
REVOKE INTEGRATED LOGIN FROM Administrator;
```

Disallow the Finance group from executing the procedure ShowCustomers.

```
REVOKE EXECUTE ON ShowCustomers FROM Finance;
```

Drop the user ID FranW from the database.

```
REVOKE CONNECT FROM FranW;
```


REVOKE CONSOLIDATE statement [SQL Remote]

Use this statement to stop a consolidated database from receiving SQL Remote messages from this database.

Syntax

REVOKE CONSOLIDATE FROM *userid*

Remarks

CONSOLIDATE permissions must be granted at a remote database for the user ID representing the consolidated database. The REVOKE CONSOLIDATE statement removes the consolidated database user ID from the list of users receiving messages from the current database.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 639
- "REVOKE REMOTE statement [SQL Remote]" on page 640
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 641
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 553

Example

- The following statement revokes consolidated status from the condb user ID:

```
REVOKE CONSOLIDATE FROM condb;
```

REVOKE PUBLISH statement [SQL Remote]

Use this statement to terminate the identification of the named user ID as the CURRENT publisher.

Syntax

REVOKE PUBLISH FROM *userid*

Remarks

Each database in a SQL Remote installation is identified in outgoing messages by a publisher user ID. The current publisher user ID can be found using the CURRENT PUBLISHER special constant. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER;
```

The REVOKE PUBLISH statement ends the identification of the named user ID as the publisher.

You should not REVOKE PUBLISH from a database while the database has active SQL Remote publications or subscriptions.

Issuing a REVOKE PUBLISH statement at a database has several consequences for a SQL Remote installation:

- You will not be able to insert data into any tables with a CURRENT PUBLISHER column as part of the primary key. Any outgoing messages will not be identified with a publisher user ID, and so will not be accepted by recipient databases.

If you change the publisher user ID at any consolidated or remote database in a SQL Remote installation, you must ensure that the new publisher user ID is granted REMOTE permissions on all databases receiving messages from the database. This will generally require all subscriptions to be dropped and recreated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “GRANT PUBLISH statement [SQL Remote]” on page 555
- “REVOKE REMOTE statement [SQL Remote]” on page 640
- “REVOKE REMOTE DBA statement [SQL Remote]” on page 641
- “REVOKE CONSOLIDATE statement [SQL Remote]” on page 638

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
REVOKE PUBLISH FROM publisher_ID;
```


REVOKE REMOTE statement [SQL Remote]

Use this statement to stop a user from being able to receive SQL Remote messages from this database.

Syntax

REVOKE REMOTE FROM *userid*, ...

Remarks

REMOTE permissions are required for a user ID to receive messages in a SQL Remote replication installation. The REVOKE REMOTE statement removes a user ID from the list of users receiving messages from the current database.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 639
- "GRANT REMOTE statement [SQL Remote]" on page 556
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 641
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 638

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
REVOKE REMOTE FROM SamS;
```


REVOKE REMOTE DBA statement [SQL Remote]

Use this statement to provide DBA privileges to a user ID, but only when connected from the Message Agent.

Syntax 1

REVOKE REMOTE DBA
FROM *userid*, …

Remarks

In MobiLink, REMOTE DBA authority is a level of permission required by the SQL Anywhere synchronization client (dbmlsync).

In SQL Remote, REMOTE DBA authority enables the Message Agent to have full access to the database to make any changes contained in the messages, while avoiding security problems associated with distributing DBA user IDs passwords.

- This statement revokes REMOTE DBA authority from a user ID.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 639
- "REVOKE REMOTE statement [SQL Remote]" on page 640
- "GRANT REMOTE DBA statement [MobiLink] [SQL Remote]" on page 558
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 638
- "Initiating synchronization" [*MobiLink - Client Administration*]
- "The Message Agent and replication security" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

ROLLBACK statement

Use this statement to end a transaction and undo any changes made since the last COMMIT or ROLLBACK.

Syntax

ROLLBACK [**WORK**]

Remarks

A transaction is the logical unit of work done on one database connection to a database between COMMIT or ROLLBACK statements. The ROLLBACK statement ends the current transaction and undoes all changes made to the database since the previous COMMIT or ROLLBACK.

Permissions

None.

Side effects

Closes all cursors not opened WITH HOLD.

See also

- "COMMIT statement" on page 367
- "ROLLBACK TO SAVEPOINT statement" on page 643

Standards and compatibility

- **SQL/2003** Core feature.

ROLLBACK TO SAVEPOINT statement

To cancel any changes made since a SAVEPOINT.

Syntax

ROLLBACK TO SAVEPOINT [*savepoint-name*]

Remarks

The ROLLBACK TO SAVEPOINT statement will undo any changes that have been made since the SAVEPOINT was established. Changes made prior to the SAVEPOINT are not undone; they are still pending.

The *savepoint-name* is an identifier that was specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is used. Any savepoints since the named savepoint are automatically released.

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

- "BEGIN TRANSACTION statement [T-SQL]" on page 354
- "COMMIT statement" on page 367
- "RELEASE SAVEPOINT statement" on page 625
- "ROLLBACK statement" on page 642
- "SAVEPOINT statement" on page 647
- "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

ROLLBACK TRANSACTION statement [T-SQL]

Use this statement to cancel any changes made since a SAVE TRANSACTION.

Syntax

ROLLBACK TRANSACTION [*savepoint-name*]

Remarks

The ROLLBACK TRANSACTION statement undoes any changes that have been made since a savepoint was established using SAVE TRANSACTION. Changes made prior to the SAVE TRANSACTION are not undone; they are still pending.

The *savepoint-name* is an identifier that was specified on a SAVE TRANSACTION statement within the current transaction. If *savepoint-name* is omitted, all outstanding changes are rolled back. Any savepoints since the named savepoint are automatically released.

Permissions

There must be a corresponding SAVE TRANSACTION within the current transaction.

Side effects

None.

See also

- "ROLLBACK TO SAVEPOINT statement" on page 643
- "BEGIN TRANSACTION statement [T-SQL]" on page 354
- "COMMIT statement" on page 367,
- "SAVE TRANSACTION statement [T-SQL]" on page 646

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example displays five rows with values 10, 20, and so on. The effect of the DELETE, but not the prior INSERTs or UPDATE, is undone by the ROLLBACK TRANSACTION statement.

```
BEGIN
    SELECT row_num INTO #tmp
    FROM sa_rowgenerator( 1, 5 )
    UPDATE #tmp SET row_num=row_num*10
    SAVE TRANSACTION before_delete
    DELETE FROM #tmp WHERE row_num >= 3
    ROLLBACK TRANSACTION before_delete
    SELECT * FROM #tmp
END
```


ROLLBACK TRIGGER statement

Use this statement to undo any changes made in a trigger.

Syntax

ROLLBACK TRIGGER [**WITH** *raiserror-statement*]

Remarks

The ROLLBACK TRIGGER statement rolls back the work done in a trigger, including the data modification that caused the trigger to fire.

Optionally, a RAISERROR statement can be issued. If a RAISERROR statement is issued, an error is returned to the application. If no RAISERROR statement is issued, no error is returned.

If a ROLLBACK TRIGGER statement is used within a nested trigger and without a RAISERROR statement, only the innermost trigger and the statement which caused it to fire are undone.

Permissions

None.

Side effects

None

See also

- "CREATE TRIGGER statement" on page 462
- "ROLLBACK statement" on page 642
- "ROLLBACK TO SAVEPOINT statement" on page 643
- "RAISERROR statement [T-SQL]" on page 616

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

SAVE TRANSACTION statement [T-SQL]

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVE TRANSACTION *savepoint-name*

Remarks

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in a ROLLBACK TRANSACTION statement. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*].

Permissions

None.

Side effects

None.

See also

- "SAVEPOINT statement" on page 647
- "BEGIN TRANSACTION statement [T-SQL]" on page 354
- "COMMIT statement" on page 367
- "ROLLBACK TRANSACTION statement [T-SQL]" on page 644

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example displays five rows with values 10, 20, and so on. The effect of the DELETE, but not the prior INSERTs or UPDATE, is undone by the ROLLBACK TRANSACTION statement.

```
BEGIN
    SELECT row_num INTO #tmp
    FROM sa_rowgenerator( 1, 5 )
    UPDATE #tmp SET row_num=row_num*10
    SAVE TRANSACTION before_delete
    DELETE FROM #tmp WHERE row_num >= 3
    ROLLBACK TRANSACTION before_delete
    SELECT * FROM #tmp
END
```


SAVEPOINT statement

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVEPOINT [*savepoint-name*]

Remarks

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in a RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT statement. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*].

Savepoints that are established while a trigger or atomic compound statement is executing are automatically released when the atomic operation ends.

You cannot modify data in a proxy table from within a savepoint.

Permissions

None.

Side effects

None.

See also

- "RELEASE SAVEPOINT statement" on page 625
- "ROLLBACK TO SAVEPOINT statement" on page 643

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

SELECT statement

Use this statement to retrieve information from the database.

Syntax

[**WITH** *temporary-views*]
  SELECT [**ALL** | **DISTINCT**] [*row-limitation*] *select-list*
[**INTO** { *hostvar-list* | *variable-list* | *table-name* }]
[**FROM** *from-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *group-by-expression*]
[**HAVING** *search-condition*]
[**WINDOW** *window-expression*]
[**ORDER BY** { *expression* | *integer* } [**ASC** | **DESC**], …]
[**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

temporary-views :
  regular-view, …
| **RECURSIVE** { *regular-view* | *recursive-view* }, …

regular-view :
  view-name [**(** *column-name*, … **)**]
  AS (*subquery* **)**

recursive-view :
  view-name **(** *column-name*, … **)**
  AS (*initial-subquery* **UNION ALL** *recursive-subquery* **)**

row-limitation :
  FIRST | **TOP** *n* [**START AT** *m*]

select-list :
expression [[**AS**] *alias-name*], …
| ***
| *window-function* **OVER** { *window-name* | *window-spec* }
  [[**AS**] *alias-name*]

from-expression :

See "FROM clause" on page 535.

group-by-expression :

See "GROUP BY clause" on page 559.

search-condition :

See "Search conditions" on page 20.

window-name : *identifier*

window-expression :

See "WINDOW clause" on page 719.

window-spec :

See "WINDOW clause" on page 719.

window-function :
RANK ( )
| **DENSE_RANK ( )**
| **PERCENT_RANK ( )**
| **CUME_DIST ( )**
| **ROW_NUMBER ( )**
| *aggregate-function*

cursor-concurrency :
BY { **VALUES** | **TIMESTAMP** | **LOCK** }

xml-mode :
RAW [, **ELEMENTS**] | **AUTO** [, **ELEMENTS**] | **EXPLICIT**

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

WITH or WITH RECURSIVE clause Define one or more common table expressions, also known as temporary views, to be used elsewhere in the remainder of the statement. These expressions may be non-recursive, or may be self-recursive. Recursive common table expressions may appear alone, or intermixed with non-recursive expressions, only if the RECURSIVE keyword is specified. Mutually recursive common table expressions are not supported.

This clause is permitted only if the SELECT statement appears in one of the following locations:

- Within a top-level SELECT statement
- Within the top-level SELECT statement of a VIEW definition
- Within a top-level SELECT statement within an INSERT statement

Recursive expressions consist of an initial subquery and a recursive subquery. The initial-query implicitly defines the schema of the view. The recursive subquery must contain a reference to the view within the FROM clause. During each iteration, this reference refers only to the rows added to the view in the previous

iteration. The reference must not appear on the null-supplying side of an outer join. A recursive common table expression must not use aggregate functions and must not contain a GROUP BY, ORDER BY, or DISTINCT clause.

The WITH clause is not supported with remote tables.

See "Common Table Expressions" [*SQL Anywhere Server - SQL Usage*].

ALL or DISTINCT clause ALL (the default) returns all rows that satisfy the clauses of the SELECT statement. If DISTINCT is specified, duplicate output rows are eliminated. Many statements take significantly longer to execute when DISTINCT is specified, so you should reserve DISTINCT for cases where it is necessary.

row-limitation clause Explicitly limit the rows of queries that include ORDER BY clauses. The TOP value must be an integer constant or integer variable with value greater than or equal to 0. The START AT value must be an integer constant or integer variable with a value greater than 0.

For more information about the use of FIRST and TOP, see "Explicitly limiting the number of rows returned by a query" [*SQL Anywhere Server - SQL Usage*].

select-list clause The *select-list* is a list of expressions, separated by commas, specifying what is retrieved from the database. An asterisk (*) means select all columns of all tables in the FROM clause.

Aggregate functions are allowed in the *select-list* (see "SQL Functions" on page 91). Subqueries are also allowed in the *select-list* (see "Expressions" on page 15). Each subquery must be within parentheses.

Alias names can be used throughout the query to represent the aliased expression.

Alias names are also displayed by Interactive SQL at the top of each column of output from the SELECT statement. If the optional alias name is not specified after an expression, Interactive SQL will display the expression itself.

INTO hostvar-list clause This clause is used in embedded SQL only. It specifies where the results of the SELECT statement will go. There must be one host variable item for each item in the *select-list*. *select-list* items are put into the host variables in order. An indicator host variable is also allowed with each host variable, so the program can tell if the *select-list* item was NULL.

INTO variable-list clause This clause is used in procedures and triggers only. It specifies where the results of the SELECT statement will go. There must be one variable for each item in the *select-list*. *select-list* items are put into the variables in order.

INTO table-name clause This clause is used to create a table and fill it with data.

If the table name starts #, it is created as a temporary table. Otherwise, the table is created as a permanent base table. For permanent tables to be created, the query must satisfy one of the following conditions:

- The *select-list* contains more than one item, and the INTO target is a single *table-name* identifier.
- The *select-list* contains a * and the INTO target is specified as *owner.table*.

To create a permanent table with one column, the table name must be specified as *owner.table*.

This statement causes a COMMIT before execution as a side effect of creating the table. RESOURCE authority is required to execute this statement. No permissions are granted on the new table: the statement is a short form for CREATE TABLE followed by INSERT ... SELECT.

Tables created using this clause do not have a primary key defined. You can add a primary key using ALTER TABLE. A primary key should be added before applying any UPDATEs or DELETEs to the table; otherwise, these operations result in all column values being logged in the transaction log for the affected rows.

FROM clause Rows are retrieved from the tables and views specified in the *table-expression*. A SELECT statement with no FROM clause can be used to display the values of expressions not derived from tables. For example, these two statements are equivalent and display the value of the global variable @@version.

```
SELECT @@version;
SELECT @@version FROM DUMMY;
```

See "FROM clause" on page 535.

WHERE clause This clause specifies which rows are selected from the tables named in the FROM clause. It can be used to do joins between multiple tables, as an alternative to the ON phrase (which is part of the FROM clause). See "Search conditions" on page 20 and "FROM clause" on page 535.

GROUP BY clause You can group by columns, alias names, or functions. The result of the query contains one row for each distinct set of values in the named columns, aliases, or functions. As with DISTINCT and the set operations UNION, INTERSECT, and EXCEPT, the GROUP BY clause treats NULL values in the same manner as any other value in each domain. In other words, multiple NULL values in a grouping attribute will form a single group. Aggregate functions can then be applied to these groups to get meaningful results.

When GROUP BY is used, the *select-list*, HAVING clause, and ORDER BY clause must not reference any identifier that is not named in the GROUP BY clause. The exception is that the *select-list* and HAVING clause may contain aggregate functions.

HAVING clause This clause selects rows based on the group values and not on the individual row values. The HAVING clause can only be used if either the statement has a GROUP BY clause or the *select-list* consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.

WINDOW clause This clause defines all or part of a window for use with window functions such as AVG and RANK. See "WINDOW clause" on page 719.

ORDER BY clause This clause sorts the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order. If the expression is an integer *n*, then the query results are sorted by the *n*th item in the *select-list*.

The only way to ensure that rows are returned in a particular order is to use ORDER BY. In the absence of an ORDER BY clause, SQL Anywhere returns rows in whatever order is most efficient. This means that the appearance of result sets may vary depending on when you last accessed the row and other factors.

In embedded SQL, the SELECT statement is used for retrieving results from the database and placing the values into host variables via the INTO clause. The SELECT statement must return only one row. For multiple row queries, you must use cursors.

FOR UPDATE or FOR READ ONLY clause These clauses specify whether updates are allowed through a cursor opened on the query, and if so, what concurrency semantics will be used. This clause cannot be used with the FOR XML clause.

When you specify FOR UPDATE BY TIMESTAMP or FOR UPDATE BY VALUES, the database server uses optimistic concurrency by using a keyset-driven cursor. In this situation, lost updates can occur.

If you do not use a FOR clause in the SELECT statement, the updatability of a cursor depends on the cursor's declaration (see "DECLARE statement" on page 477 and "FOR statement" on page 530) and how cursor concurrency is specified by the API. In ODBC, JDBC, and OLE DB, statement updatability is explicit and a read-only, forward-only cursor is used unless it is overridden by the application. In Open Client, embedded SQL, and within stored procedures, cursor updatability does not have to be specified, and the default is FOR UPDATE.

To ensure that a statement acquires an intent lock, you must do one of the following:

- specify FOR UPDATE BY LOCK in the query
- specify HOLDLOCK, WITH (HOLDLOCK), WITH (UPDLOCK), or WITH (XLOCK) in the FROM clause of the query
- open the cursor with API calls that specify CONCUR_LOCK
- fetch the rows with attributes indicating fetch for update

In addition to cursor updatability, statement updatability is also dependent on the setting of the ansi_update_constraints database option and the specific characteristics of the statement, including whether the statement contains ORDER BY, DISTINCT, GROUP BY, HAVING, UNION, aggregate functions, joins, or non-updatable views.

For more information about cursor sensitivity, see "SQL Anywhere cursors" [*SQL Anywhere Server - Programming*].

For more information about ODBC concurrency, see the discussion of SQLSetStmtAttr in "Choosing ODBC cursor characteristics" [*SQL Anywhere Server - Programming*].

For more information about the ansi_update_constraints database option, see "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

For more information about cursor updatability, see "Understanding updatable statements" [*SQL Anywhere Server - Programming*].

FOR XML clause This clause specifies that the result set is to be returned as an XML document. The format of the XML depends on the mode you specify. This clause cannot be used with the FOR UPDATE or FOR READ ONLY clause.

When you specify RAW mode, each row in the result set is represented as an XML <row> element, and each column is represented as an attribute of the <row> element.

AUTO mode returns the query results as nested XML elements. Each table referenced in the *select-list* is represented as an element in the XML. The order of nesting for the elements is based on the order that tables are referenced in the *select-list*.

EXPLICIT mode allows you to control the form of the generated XML document. Using EXPLICIT mode offers more flexibility in naming elements and specifying the nesting structure than either RAW or AUTO mode. See "Using FOR XML EXPLICIT" [*SQL Anywhere Server - SQL Usage*].

For more information about using the FOR XML clause, see "Using the FOR XML clause to retrieve query results as XML" [*SQL Anywhere Server - SQL Usage*].

OPTION clause

This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

  For more information about simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].
- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:
  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The SELECT statement is used for retrieving results from the database.

A SELECT statement can be used in Interactive SQL to browse data in the database, or to export data from the database to an external file.

A SELECT statement can also be used in procedures and triggers or in embedded SQL. A SELECT statement with an INTO clause is used for retrieving results from the database when the SELECT statement only returns one row. For multiple row queries, you must use cursors.

A SELECT statement can also be used to return a result set from a procedure.

Note
When a GROUP BY expression is used in a SELECT statement, the *select-list*, HAVING clause, and ORDER BY clause can reference only identifiers named in the GROUP BY clause. The exception is that the *select-list* and HAVING clause may contain aggregate functions.

Permissions

Must have SELECT permission on the named tables and views.

Side effects

None.

See also

- "Expressions" on page 15
- "FROM clause" on page 535
- "Search conditions" on page 20
- "UNION statement" on page 695
- "Joins: Retrieving Data from Several Tables" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature. The complexity of the SELECT statement means that you should check individual clauses against the standard. For example, the ROLLUP keyword is part of feature T431.

  FOR UPDATE, FOR READ ONLY, and FOR UPDATE (*column-list*) are core features.

  FOR UPDATE BY [LOCK | TIMESTAMP | VALUES] is a SQL Anywhere vendor extension.

Example

This example returns the total number of employees in the Employees table.

```
SELECT COUNT(*)
FROM Employees;
```

This example lists all customers and the total value of their orders.

```
SELECT CompanyName,
   CAST( SUM( SalesOrderItems.Quantity *
   Products.UnitPrice ) AS INTEGER ) VALUE
FROM Customers
   JOIN SalesOrders
   JOIN SalesOrderItems
   JOIN Products
GROUP BY CompanyName
ORDER BY VALUE DESC;
```

The following statement shows an embedded SQL SELECT statement:


```
SELECT count(*) INTO :size
FROM Employees;
```

The following statement is optimized to return the first row in the result set quickly:

```
SELECT Name
FROM Products
GROUP BY Name
HAVING COUNT( * ) > 1
AND MAX( UnitPrice ) > 10
OPTION( optimization_goal = 'first-row' );
```


SET statement

Use this statement to assign a value to a SQL variable.

Syntax

SET *identifier = expression*

Remarks

The SET statement assigns a new value to a variable. The variable must have been previously created using a CREATE VARIABLE statement or DECLARE statement, or it must be an OUTPUT parameter for a procedure. The variable name can optionally use the Transact-SQL convention of an @ sign preceding the name. For example,

```
SET @localvar = 42
```

A variable can be used in a SQL statement anywhere a column name is allowed. If a column name exists with the same name as the variable, the variable value is used.

Variables are local to the current connection, and disappear when you disconnect from the database or use the DROP VARIABLE statement. They are not affected by COMMIT or ROLLBACK statements.

Variables are necessary for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs because embedded SQL host variables are limited to 32,767 bytes.

Permissions

None.

Side effects

None.

See also

- "CREATE VARIABLE statement" on page 469
- "DECLARE statement" on page 477
- "DROP VARIABLE statement" on page 512
- "Expressions" on page 15

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

This simple example shows the creation of a variable called 'birthday', and uses SET to set the date to CURRENT DATE.

```
CREATE VARIABLE @birthday DATE;
    SET @birthday = CURRENT DATE;
```

The following code fragment inserts a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_VARCHAR( 500 ) buffer;
```



```
/* Note: maximum DECL_VARCHAR size is 32765 */
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG VARCHAR;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( buffer, 1, 5000, fp );
   if( size <= 0 ) break;
   /* Does not work if data contains null chars */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES( 1, hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```

The following code fragment inserts a large binary value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY( 5000 ) buffer;
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG BINARY;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( &(buffer.array), 1, 5000, fp );
   if( size <= 0 ) break;
   buffer.len = size;
   /* add data to blob using concatenation */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```


SET statement [T-SQL]

Use this statement to set database options for the current connection in an Adaptive Server Enterprise-compatible manner.

Syntax

SET *option-name option-value*

Remarks

The available options are as follows:

Option name	Option value
ansinull	On or Off
ansi_permissions	On or Off
close_on_endtrans	On or Off
datefirst	1, 2, 3, 4, 5, 6, or 7 The setting of this option affects the DATEPART function when obtaining a weekday value. For more information about specifying the first day of the week, see "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*] and "DATEPART function [Date and time]" on page 140.
quoted_identifier	On | Off
rowcount	*integer*
self_recursion	On | Off
string_rtruncation	On | Off
textsize	*integer*
transaction isolation level	0, 1, 2, 3, snapshot, statement snapshot, or read only statement snapshot

Database options in SQL Anywhere are set using the SET OPTION statement. However, SQL Anywhere also provides support for the Adaptive Server Enterprise SET statement for options that are particularly useful for compatibility.

The following options can be set using the Transact-SQL SET statement in SQL Anywhere, as well as in Adaptive Server Enterprise:

- **SET ansinull { On | Off }**
  The default behavior for comparing values to NULL is different in SQL Anywhere and Adaptive Server Enterprise. Setting ansinull to Off provides Transact-SQL compatible comparisons with NULL.

  SQL Anywhere also supports the following syntax:

  SET ansi_nulls { **On** | **Off** }

  For more information, see "ansinull option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET ansi_permissions { On | Off }** The default behavior is different in SQL Anywhere and Adaptive Server Enterprise regarding permissions required to carry out an UPDATE or DELETE containing a column reference. Setting ansi_permissions to Off provides Transact-SQL-compatible permissions on UPDATE and DELETE. See "ansi_permissions option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET close_on_endtrans { On | Off }** The default behavior is different in SQL Anywhere and Adaptive Server Enterprise for closing cursors at the end of a transaction. Setting close_on_endtrans to Off provides Transact-SQL compatible behavior. See "close_on_endtrans option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET datefirst { 1 | 2 | 3 | 4 | 5 | 6 | 7 }** The default is 7, which means that the first day of the week is by default Sunday. To set this option permanently, see "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*].

- **SET quoted_identifier { On | Off }** Controls whether strings enclosed in double quotes are interpreted as identifiers (On) or as literal strings (Off). See "Setting options for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Usage*] and "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET rowcount** *integer* The Transact-SQL ROWCOUNT option limits the number of rows fetched for any cursor to the specified integer. This includes rows fetched by re-positioning the cursor. Any fetches beyond this maximum return a warning. The option setting is considered when returning the estimate of the number of rows for a cursor on an OPEN request.

  SET ROWCOUNT also limits the number of rows affected by a searched UPDATE or DELETE statement to *integer*. This might be used, for example, to allow COMMIT statements to be performed at regular intervals to limit the size of the rollback log and lock table. The application (or procedure) would need to provide a loop to cause the update/delete to be re-issued for rows that are not affected by the first operation. A simple example is given below:

```
BEGIN
   DECLARE @count INTEGER
   SET rowcount 20
   WHILE(1=1) BEGIN
      UPDATE Employees SET Surname='new_name'
      WHERE Surname <> 'old_name'
      /* Stop when no rows changed */
      SELECT @count = @@rowcount
      IF @count = 0 BREAK
      PRINT string('Updated ',
               @count,' rows; repeating...')
```



```
        COMMIT
    END
    SET rowcount 0
END
```

In SQL Anywhere, if the ROWCOUNT setting is greater than the number of rows that Interactive SQL can display, Interactive SQL may do some extra fetches to reposition the cursor. Thus, the number of rows actually displayed may be less than the number requested. Also, if any rows are re-fetched due to truncation warnings, the count may be inaccurate.

A value of zero resets the option to get all rows.

- **SET self_recursion { On | Off }** The self_recursion option is used within triggers to enable (On) or prevent (Off) operations on the table associated with the trigger from firing other triggers.
- **SET string_rtruncation { On | Off }** The default behavior is different between SQL Anywhere and Adaptive Server Enterprise when non-space characters are truncated during assignment of SQL string data. Setting string_rtruncation to On provides Transact-SQL-compatible string comparisons. See "string_rtruncation option [compatibility]" [*SQL Anywhere Server - Database Administration*].
- **SET textsize** Specifies the maximum size (in bytes) of text or image type data to be returned with a select statement. The @@textsize global variable stores the current setting. To reset to the default size (32 KB), use the command:

  ```
  set textsize 0
  ```

- **SET transaction isolation level { 0 | 1 | 2 | 3 | snapshot | statement snapshot | read only statement snapshot }** Sets the locking isolation level for the current connection, as described in "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*]. For Adaptive Server Enterprise, only 1 and 3 are valid options. For SQL Anywhere, any of 0, 1, 2, 3, snapshot, statement snapshot, and read only statement snapshot is a valid option. See "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*].

The SET statement is allowed by SQL Anywhere for the prefetch option, for compatibility, but has no effect.

Permissions

None.

Side effects

None.

See also

- "SET OPTION statement" on page 664
- "Setting options for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Usage*]
- "Compatibility options" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

SET CONNECTION statement [Interactive SQL] [ESQL]

Use this statement to change the active database connection.

Syntax

SET CONNECTION [*connection-name*]

connection-name : *identifier*, *string*, or *hostvar*

Remarks

The SET CONNECTION statement changes the active database connection to connection-name. The current connection state is saved, and is resumed when it again becomes the active connection. If connection-name is omitted and there is a connection that was not named, that connection becomes the active connection.

When cursors are opened in embedded SQL, they are associated with the current connection. When the connection is changed, the cursor names of the previously active connection become inaccessible. These cursors remain active and in position, and become accessible when the associated connection becomes active again.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 370
- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 497

Standards and compatibility

- **SQL/2003** Interactive SQL is a vendor extension. Embedded SQL is a core feature.

Example

The following example is in embedded SQL.

```
EXEC SQL SET CONNECTION :conn_name;
```

From Interactive SQL, set the current connection to the connection named conn1.

```
SET CONNECTION conn1;
```


SET DESCRIPTOR statement [ESQL]

Use this statement to describe the variables in a SQL descriptor area and to place data into the descriptor area.

Syntax

SET DESCRIPTOR *descriptor-name*
{ **COUNT** = { *integer* | *hostvar* }
| **VALUE** { *integer* | *hostvar* } *assignment*, … }

assignment :
{ **TYPE** | **SCALE** | **PRECISION** | **LENGTH** | **INDICATOR** }
= { *integer* | *hostvar* }
| **DATA** = *hostvar*

Remarks

The SET DESCRIPTOR statement is used to describe the variables in a descriptor area, and to place data into the descriptor area.

The SET … COUNT statement sets the number of described variables within the descriptor area. The value for count must not exceed the number of variables specified when the descriptor area was allocated.

The value { *integer* | *hostvar* } specifies the variable in the descriptor area upon which the assignment(s) is performed.

Type checking is performed when doing SET … DATA, to ensure that the variable in the descriptor area has the same type as the host variable. LONG VARCHAR and LONG BINARY are not supported by SET DESCRIPTOR ... DATA.

If an error occurs, the code is returned in the SQLCA.

Permissions

None.

Side effects

None.

See also

- “ALLOCATE DESCRIPTOR statement [ESQL]” on page 299
- “DEALLOCATE DESCRIPTOR statement [ESQL]” on page 475
- “The SQL descriptor area (SQLDA)” [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside of core SQL.

Example

The following example sets the type of the column with position col_num in sqlda.

```
VOID set_type( SQLDA *sqlda, INT col_num, INT new_type )
{
```



```
    EXEC SQL BEGIN DECLARE SECTION;
    INT new_type1 = new_type;
    INT col = col_num;
    EXEC SQL END DECLARE SECTION;
EXEC SQL SET DESCRIPTOR sqlda VALUE :col TYPE = :new_type1;
}
```

For a longer example, see “ALLOCATE DESCRIPTOR statement [ESQL]” on page 299.

SET OPTION statement

Use this statement to change the values of database options.

Syntax

SET [**EXISTING**] [**TEMPORARY**] **OPTION**
[*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Embedded SQL syntax

SET [**TEMPORARY**] **OPTION**
[*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*, *string*, or *hostvar*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Remarks

The SET OPTION statement is used to change options that affect the behavior of the database server. Setting the value of an option can change the behavior for all users or only for an individual user. The scope of the change can be either temporary or permanent.

Any option, whether user-defined or not, must have a public setting before a user-specific value can be assigned. The database server does not support setting TEMPORARY values for user-defined options.

The classes of options are:

- General database options
- Transact-SQL compatibility
- Replication database options

For a listing and description of all available options, see "Database Options" [*SQL Anywhere Server - Database Administration*].

You can set options at three levels of scope: public, user, and temporary. A temporary option takes precedence over other options, and user options take precedence over public options. If you set a user level option for the current user, the corresponding temporary option gets set as well.

If you use the EXISTING keyword, option values cannot be set for an individual user ID unless there is already a PUBLIC user ID setting for that option.

If you specify a user ID, the option value applies to that user (or, for a group user ID, the members of that group). If you specify PUBLIC, the option value applies to all users who do not have an individual setting for the option. By default, the option value applies to the currently logged on user ID that issued the SET OPTION statement.

For example, the following statement applies an option change to the user DBA, if DBA is the user issuing the SQL statement:

```
SET OPTION precision = 40;
```

However the following statement applies the change to the PUBLIC user ID, a user group to which all users belong.

```
SET OPTION Public.login_mode = Standard;
```

Only users with DBA privileges have the authority to set an option for the PUBLIC user ID.

Users can use the SET OPTION statement to change the values for their own user ID. Setting the value of an option for a user ID other then your own is permitted only if you have DBA authority.

Adding the TEMPORARY keyword to the SET OPTION statement changes the duration that the change takes effect. By default, the option value is permanent: it will not change until it is explicitly changed using the SET OPTION statement.

When the SET TEMPORARY OPTION statement is not qualified with a user ID, the new option value is in effect only for the current connection.

When SET TEMPORARY OPTION is used for the PUBLIC user ID, the change is in place for as long as the database is running. When the database is shut down, TEMPORARY options for the PUBLIC group revert back to their permanent value.

Setting temporary options for the PUBLIC user ID offers a security benefit. For example, when the login_mode option is enabled, the database relies on the login security of the system on which it is running. Enabling it temporarily means that a database relying on the security of a Windows domain will not be compromised if the database is shut down and copied to a local computer. In that case, the temporary enabling of the login_mode option reverts to its permanent value, which could be Standard, a mode where integrated logins are not permitted.

If *option-value* is omitted, the specified option setting is deleted from the database. If it was a personal option setting, the value will revert back to the PUBLIC setting. If a TEMPORARY option is deleted, the option setting will revert back to the permanent setting.

Caution
Changing option settings while fetching rows from a cursor is not supported, as it can lead to ill-defined behavior. For example, changing the date_format setting while fetching from a cursor would lead to different date formats among the rows in the result set. Do not change option settings while fetching rows.

The SET OPTION statement is ignored by the SQL Flagger.

Permissions

None required to set your own options.

DBA authority is required to set database options for another user or PUBLIC.

Side effects

If TEMPORARY is not specified, an automatic commit is performed.

See also

- "Database options" [*SQL Anywhere Server - Database Administration*]
- "Compatibility options" [*SQL Anywhere Server - Database Administration*]
- "SQL Remote options" [*SQL Anywhere Server - Database Administration*]
- "SET OPTION statement [Interactive SQL]" on page 667

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Set the date format option to on:

```
SET OPTION public.date_format = 'Mmm dd yyyy';
```

Set the date format option to its default setting:

```
SET OPTION public.date_format =;
```

Set the wait_for_commit option to On:

```
SET OPTION wait_for_commit = 'On';
```

Following are two embedded SQL examples.

```
1. EXEC SQL SET OPTION :user.:option_name = :value;
2. EXEC SQL SET TEMPORARY OPTION date_format = 'mm/dd/yyyy';
```


SET OPTION statement [Interactive SQL]

Use this statement to change the values of Interactive SQL options.

Syntax 1

SET [**TEMPORARY**] **OPTION** *option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*, *string*, or *hostvar*

option-value : *string*, *identifier*, or *number*

Syntax 2

SET PERMANENT

Syntax 3

SET

Remarks

Syntax 1 stores the specified Interactive SQL option.

Syntax 2 stores all current Interactive SQL options

Syntax 3 displays all of the current database option settings. If there are temporary options set for the database server, these are displayed; otherwise, the permanent option settings are displayed.

Interactive SQL option settings are stored on the client computer. They are not stored in the database.

> **Note**
> The syntax SET [TEMPORARY] OPTION [*userid* | PUBLIC.]*option-name* is deprecated. If you specify this syntax, the *userid* or PUBLIC keyword is ignored.

See also

- "Interactive SQL options" [*SQL Anywhere Server - Database Administration*]

SET REMOTE OPTION statement [SQL Remote]

Use this statement to set a message control parameter for a SQL Remote message link.

Syntax

SET REMOTE *link-name* **OPTION**
[*userid*.| **PUBLIC**.]*link-option-name* = *link-option-value*

link-name:
file | **ftp** | **mapi** | **smtp** | **vim**

link-option-name:
common-option | *file-option* | *ftp-option*
| *mapi-option* | *smtp-option* | *vim-option*

common-option:
debug | **output_log_send_on_error**
| **output_log_send_limit** | **output_log_send_now**

file-option:
directory | **invalid_extensions** | **unlink_delay**

ftp-option:
active_mode | **host** | **invalid_extensions** | **password** | **port** | **root_directory** | **user** | **reconnect_retries** | **reconnect_pause**

mapi-option:
force_download | **ipm_receive** | **ipm_send** | **profile**

smtp-option:
local_host | **pop3_host** | **pop3_password** | **pop3_userid**
| **smtp_host** | **top_supported**

vim-option:
password | **path** | **receive_all** | **send_vim_mail** | **userid**

link-option-value : *string*

Parameters

userid If no *userid* is specified, then the current publisher is assumed.

Option values The option values are message-link dependent. For more information, see:

- "The file message system" [*SQL Remote*]
- "The ftp message system" [*SQL Remote*]
- "The MAPI message system" [*SQL Remote*]
- "The SMTP message system" [*SQL Remote*]
- "The VIM message system" [*SQL Remote*]

Remarks

The Message Agent saves message link parameters when the user enters them in the message link dialog box when the message link is first used. In this case, it is not necessary to use this statement explicitly. This statement is most useful when preparing a consolidated database for extracting many databases.

The option names are case sensitive. The case sensitivity of option values depends on the option: Boolean values are case insensitive, while the case sensitivity of passwords, directory names, and other strings depend on the cases sensitivity of the file system (for directory names), or the database (for user IDs and passwords).

Note
Support for VIM and MAPI is deprecated.

Permissions

Must have DBA authority. The publisher can set their own options.

Side effects

Automatic commit.

See also

- "Troubleshooting errors at remote sites" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement sets the FTP host to *ftp.mycompany.com* for the ftp link for user myuser:

```
SET REMOTE FTP OPTION myuser.host = 'ftp.mycompany.com';
```

The following statement stops SQL Remote from using the specified file extensions for messages that are generated:

```
SET REMOTE ftp OPTION "Public"."invalid_extensions" =
'exe,pif,dll,bat,cmd,vbs';
```


SET SQLCA statement [ESQL]

Use this statement to tell the SQL preprocessor to use a SQLCA other than the default, global *sqlca*.

Syntax

SET SQLCA *sqlca*

sqlca : *identifier* or *string*

Remarks

The SET SQLCA statement tells the SQL preprocessor to use a SQLCA other than the default global *sqlca*. The sqlca must be an identifier or string that is a C language reference to a SQLCA pointer.

The current SQLCA pointer is implicitly passed to the database interface library on every embedded SQL statement. All embedded SQL statements that follow this statement in the C source file will use the new SQLCA.

This statement is necessary only when you are writing code that is reentrant (see "SQLCA management for multi-threaded or reentrant code" [*SQL Anywhere Server - Programming*]). The sqlca should reference a local variable. Any global or module static variable is subject to being modified by another thread.

Permissions

None.

Side effects

None.

See also

- "SQLCA management for multi-threaded or reentrant code" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The owning function could be found in a Windows DLL. Each application that uses the DLL has its own SQLCA.

```
an_sql_code FAR PASCAL ExecuteSQL( an_application *app, char *com )
{
   EXEC SQL BEGIN DECLARE SECTION;
   char *sqlcommand;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL SET SQLCA "&app->.sqlca";
   sqlcommand = com;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL EXECUTE IMMEDIATE :sqlcommand;
return( SQLCODE );
}
```


SETUSER statement

Use this statement to allow a database administrator to impersonate another user, and to enable connection pooling.

Syntax

{ **SET SESSION AUTHORIZATION** | **SETUSER** }
[[**WITH OPTION**] *userid*]

Parameters

WITH OPTION By default, only permissions (including group membership) are altered. If WITH OPTION is specified, the database options in effect are changed to the current database options of *userid*.

userid The user ID is an identifier (SETUSER syntax) or a string (SET SESSION AUTHORIZATION syntax). See "Identifiers" on page 7 and "Strings" on page 8.

Remarks

The SETUSER statement is provided to make database administration easier. It enables a user with DBA authority to impersonate another user of the database. After running a SETUSER statement, you can check which user you are impersonating by running one of the following commands:

- SELECT USER
- SELECT CURRENT USER

SETUSER can also be used from an application server to take advantage of connection pooling. Connection pooling cuts down the number of distinct connections that need to be made, which can improve performance.

SETUSER with no user ID undoes all earlier SETUSER statements.

The SETUSER statement cannot be used inside a procedure, trigger, event handler or batch.

There are several uses for the SETUSER statement, including the following:

- **Creating objects** You can use SETUSER to create a database object that is to be owned by another user.
- **Permissions checking** By acting as another user, with their permissions and group memberships, a database administrator can test the permissions and name resolution of queries, procedures, views, and so on.
- **Providing a safer environment for administrators** The database administrator has permission to carry out any action in the database. If you want to ensure that you do not accidentally carry out an unintended action, you can use SETUSER to switch to a different user ID with fewer permissions.

Note
The SETUSER statement cannot be used within procedures, triggers, events, or batches.

Permissions

Must have DBA authority.

See also

- "EXECUTE IMMEDIATE statement [SP]" on page 519
- "GRANT statement" on page 548
- "REVOKE statement" on page 636
- "SET OPTION statement" on page 664

Standards and compatibility

- **SQL/2003** SET SESSION AUTHORIZATION is a core feature. SETUSER is a vendor extension.

Example

The following statements, executed by a user named DBA, change the user ID to be Joe, then Jane, and then back to DBA.

```
SETUSER "Joe"
// ... operations...
SETUSER WITH OPTION "Jane"
// ... operations...
SETUSER
```

The following statement sets the user to Jane. The user ID is supplied as a string rather than as an identifier.

```
SET SESSION AUTHORIZATION 'Jane';
```


SIGNAL statement

Use this statement to signal an exception condition.

Syntax

SIGNAL *exception-name*

Remarks

SIGNAL allows you to raise an exception. See "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*] for a description of how exceptions are handled.

exception-name The name of an exception declared using a DECLARE statement at the beginning of the current compound statement. The exception must correspond to a system-defined SQLSTATE or a user-defined SQLSTATE. User-defined SQLSTATE values must be in the range 99000 to 99999.

Permissions

None.

Side effects

None.

See also

- "RESIGNAL statement" on page 630
- "BEGIN statement" on page 351
- "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following compound statement declares and signals a user-defined exception. If you execute this example from Interactive SQL, the message appears on the Messages tab.

```
BEGIN
   DECLARE myexception EXCEPTION
   FOR SQLSTATE '99001';
   SIGNAL myexception;
   EXCEPTION
      WHEN myexception THEN
         MESSAGE 'My exception signaled'
         TO CLIENT;
END
```


START DATABASE statement

Use this statement to start a database on the current database server.

Syntax

START DATABASE *database-file* [*start-options* ...]

start-options :
[**AS** *database-name*]
[**ON** *database-server-name*]
[**WITH TRUNCATE AT CHECKPOINT**]
[**FOR READ ONLY**]
[**AUTOSTOP** { **ON** | **OFF** }]
[**KEY** *key*]
[**WITH SERVER NAME** *alternative-database-server-name*]
[**DIRECTORY** *dbspace-directory*]

Parameters

The *start-options* can be listed in any order.

START DATABASE clause The *database-file* parameter is a string. If a relative path is supplied in *database-file*, it is relative to the database server starting directory.

AS clause If *database-name* is not specified, a default name is assigned to the database. This default name is the root of the database file. For example, a database in file *C:\Database Files\demo.db* would be given the default name of demo. The *database-name* parameter is an identifier.

ON clause This clause is supported from Interactive SQL only. In Interactive SQL, if *server-name* is not specified, the default server is the first started server among those currently running. The *server-name* parameter is an identifier.

WITH TRUNCATE AT CHECKPOINT clause Starts a database with log truncation on checkpoint enabled.

FOR READ ONLY Starts a database in read-only mode.

AUTOSTOP clause The default setting for the AUTOSTOP clause is ON. With AUTOSTOP set to ON, the database is unloaded when the last connection to it is dropped. If AUTOSTOP is set to OFF, the database is not unloaded.

In Interactive SQL, you can use YES or NO as alternatives to ON and OFF.

KEY clause If the database is strongly encrypted, enter the KEY value (password) using this clause

WITH SERVER NAME clause Use this clause to specify an alternate name for the database server when connecting to this database. If you are using database mirroring, the primary and mirror servers must both have the same database server name because clients do not know to which server they are connecting.

For more information about alternate server names and database mirroring, see "-sn database option" [*SQL Anywhere Server - Database Administration*] and "Introduction to database mirroring" [*SQL Anywhere Server - Database Administration*].

DIRECTORY clause Use this clause to specify the directory where the dbspace files are located for the database that is being started. For example, if the database server is started in the same directory as all of the dbspaces, and you include the `DIRECTORY '.'` clause, then this instructs the database server to find all dbspaces in the current directory. See "-ds database option" [*SQL Anywhere Server - Database Administration*].

Remarks

Starts a specified database on the current database server.

If you are not connected to a database and you want to use the START DATABASE statement, you must first connect to a database, such as the utility database.

For information about the utility database, see "Using the utility database" [*SQL Anywhere Server - Database Administration*].

The START DATABASE statement does not connect the current application to the specified database: an explicit connection is still needed.

Interactive SQL supports the ON clause, which allows the database to be started on a database server other than the current.

Permissions

The required permissions are specified by the database server -gd option. This option defaults to all on the personal database server, and DBA on the network server.

Side effects

None

See also

- "STOP DATABASE statement" on page 683
- "CONNECT statement [ESQL] [Interactive SQL]" on page 370
- "-gd server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start the database file *C:\Database Files\sample_2.db* on the current server.

```
START DATABASE 'c:\database files\sample_2.db';
```

From Interactive SQL, start the database file *c:\Database Files\sample_2.db* as sam2 on the server named sample.

```
START DATABASE 'c:\database files\sample_2.db'
AS sam2
ON sample;
```


START ENGINE statement [Interactive SQL]

Use this statement to start a database server.

Syntax

START ENGINE AS *database-server-name* [**STARTLINE** *command-string*]

Remarks

The START ENGINE statement starts a database server. If you want to specify a set of options for the database server, use the STARTLINE keyword together with a command string. Valid command strings are those that conform to the database server description in "The SQL Anywhere database server" [*SQL Anywhere Server - Database Administration*].

Permissions

None

Side effects

None

See also

- "STOP ENGINE statement" on page 684
- "The SQL Anywhere database server" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start a database server, named sample, without starting any databases on it.

```
START ENGINE AS sample;
```

The following example shows the use of a STARTLINE clause.

```
START ENGINE AS eng1 STARTLINE 'dbeng10 -c 8M';
```


START JAVA statement

Use this statement to start the Java VM.

Syntax

START JAVA

Remarks

The START JAVA statement starts the Java VM. The main use is to load the Java VM at a convenient time so that when the user starts to use Java functionality there is no initial pause while the Java VM is loaded.

The database server must be set up to locate a Java VM. Since you can specify different Java VMs for each database, the java_location option can be used to indicate the location (path) of the Java VM. See "java_location option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about starting the Java VM, see "Starting and stopping the Java VM" [*SQL Anywhere Server - Programming*].

Permissions

A Java VM must be installed, and the database must be Java-enabled.

This statement is not supported on Windows CE.

Side effects

None

See also

- "STOP JAVA statement" on page 685

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start the Java VM.

```
START JAVA;
```


START LOGGING statement [Interactive SQL]

Use this statement to start logging executed SQL statements to a log file.

Syntax

START LOGGING *file-name*

Remarks

The START LOGGING statement starts copying all subsequent executed SQL statements to the log file that you specify. If the file does not exist, Interactive SQL creates it. Logging continues until you explicitly stop the logging process with the STOP LOGGING statement, or until you end the current Interactive SQL session. You can also start and stop logging by clicking SQL ► Start Logging and SQL ► Stop Logging.

Permissions

None.

Side effects

None.

See also

- "STOP LOGGING statement [Interactive SQL]" on page 686

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start logging to a file called *filename.sql*, located in the c: directory.

```
START LOGGING 'c:\filename.sql';
```


START SUBSCRIPTION statement [SQL Remote]

Use this statement to start a subscription for a user to a publication.

Syntax

START SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*, …

Parameters

publication-name The name of the publication to which the user is being subscribed. This may include the owner of the publication.

subscription-value A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.

subscriber-id The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be **started** when publication updates are being sent from the consolidated database to the remote database.

The START SUBSCRIPTION statement is one of a set of statements that manage subscriptions. The CREATE SUBSCRIPTION statement defines the data that the subscriber is to receive. The SYNCHRONIZE SUBSCRIPTION statement ensures that the consolidated and remote databases are consistent with each other. The START SUBSCRIPTION statement is required to start messages being sent to the subscriber.

Data at each end of the subscription must be consistent before a subscription is started. It is recommended that you use the database extraction utility to manage the creation, synchronization, and starting of subscriptions. If you use the database extraction utility, you do not need to execute an explicit START SUBSCRIPTION statement. Also, the Message Agent starts subscriptions once they are synchronized.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 443
- "REMOTE RESET statement [SQL Remote]" on page 626
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 689
- "STOP SUBSCRIPTION statement [SQL Remote]" on page 687
- "Database Extraction utility" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement starts the subscription of user **SamS** to the **pub_contact** publication.

```
START SUBSCRIPTION TO pub_contact
FOR SamS;
```

START SYNCHRONIZATION DELETE statement [MobiLink]

Use this statement to restart logging of deletes for MobiLink synchronization.

Syntax

START SYNCHRONIZATION DELETE

Remarks

Ordinarily, SQL Anywhere and UltraLite automatically log any changes made to tables or columns that are part of a synchronization, and upload these changes to the consolidated database during the next synchronization. You can temporarily suspend automatic logging of delete operations using the STOP SYNCHRONIZATION DELETE statement. The START SYNCHRONIZATION DELETE statement allows you to restart the automatic logging.

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the delete operations executed on that connection are synchronized. The effect continues until a START SYNCHRONIZATION DELETE statement is executed. Repeating STOP SYNCHRONIZATION DELETE has no additional effect.

A single START SYNCHRONIZATION DELETE statement restarts the logging, regardless of the number of STOP SYNCHRONIZATION DELETE statements preceding it.

Do not use START SYNCHRONIZATION DELETE if your application does not synchronize data.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "STOP SYNCHRONIZATION DELETE statement [MobiLink]" on page 688
- "StartSynchronizationDelete method" [*UltraLite - .NET Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following sequence of SQL statements illustrates how to use START SYNCHRONIZATION DELETE and STOP SYNCHRONIZATION DELETE:

```
-- Prevent deletes from being sent
-- to the consolidated database
STOP SYNCHRONIZATION DELETE;

-- Remove all records older than 1 month
-- from the remote database,
-- NOT the consolidated database
DELETE FROM PROPOSAL
```



```
WHERE last_modified < months( CURRENT TIMESTAMP, -1 )

-- Re-enable all deletes to be sent
-- to the consolidated database
-- DO NOT FORGET to start this
START SYNCHRONIZATION DELETE;

-- Commit the entire operation,
-- otherwise rollback everything
-- including the stopping of the deletes
commit;
```

STOP DATABASE statement

Use this statement to stop a database on the current database server.

Syntax

STOP DATABASE *database-name*
[**ON** *database-server-name*]
[**UNCONDITIONALLY**]

Parameters

STOP DATABASE clause The *database-name* is the name of a database (other than the current database) running on the current server.

ON clause This clause is supported in Interactive SQL only. If *database-server-name* is not specified in Interactive SQL, all running servers are searched for a database of the specified name.

When not using this statement in Interactive SQL, the database is stopped only if it is started on the current database server.

UNCONDITIONALLY keyword Stop the database even if there are connections to the database. By default, the database is not stopped if there are connections to it.

Remarks

The STOP DATABASE statement stops a specified database on the current database server.

Permissions

The required permissions are specified by the database server -gk option. This option defaults to all on the personal database server, and DBA on the network server.

You cannot use STOP DATABASE on the database to which you are currently connected.

Side effects

None

See also

- "START DATABASE statement" on page 674
- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 497
- "-gd server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the database named *sample* on the current server.

```
STOP DATABASE sample;
```


STOP ENGINE statement

Use this statement to stop a database server.

Syntax

STOP ENGINE [*database-server-name*] [**UNCONDITIONALLY**]

Parameters

STOP ENGINE clause The *database-server-name* can be used in Interactive SQL only. If you are not running this statement in Interactive SQL, the current database server is stopped.

UNCONDITIONALLY keyword If you are the only connection to the database server, you do not need to use UNCONDITIONALLY. If there are other connections, the database server stops only if you use the UNCONDITIONALLY keyword.

Remarks

The STOP ENGINE statement stops the specified database server. If the UNCONDITIONALLY keyword is supplied, the database server is stopped even if there are other connections to the database server. By default, the database server will not be stopped if there are other connections to it.

The STOP ENGINE statement cannot be used in stored procedures, triggers, events, or batches.

Permissions

The permissions to shut down a server depend on the -gk setting on the database server command line. The default setting is all for the personal server, and DBA for the network server.

Side effects

None

See also

- "START ENGINE statement [Interactive SQL]" on page 676
- "-gk server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the current database server, as long as there are no other connections.

```
STOP ENGINE;
```


STOP JAVA statement

Use this statement to stop the Java VM.

Syntax

STOP JAVA

Remarks

The STOP JAVA statement unloads the Java VM when it is not in use. The main use is to economize on the use of system resources.

Permissions

This statement is not supported on Windows CE.

Side effects

None

See also

- "START JAVA statement" on page 677

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the Java VM.

```
STOP JAVA;
```


STOP LOGGING statement [Interactive SQL]

Use this statement to stop logging of SQL statements in the current session.

Syntax

STOP LOGGING

Remarks

The STOP LOGGING statement stops Interactive SQL from writing each SQL statement you execute to a log file. You can start logging with the START LOGGING statement. You can also start and stop logging by clicking SQL ► Start Logging and SQL ► Stop Logging.

Permissions

None.

Side effects

None.

See also

- "START LOGGING statement [Interactive SQL]" on page 678

Example

The following example stops the current logging session.

```
STOP LOGGING;
```


STOP SUBSCRIPTION statement [SQL Remote]

Use this statement to stop a subscription for a user to a publication.

Syntax

STOP SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*, …

Parameters

publication-name The name of the publication to which the user is being subscribed. This may include the owner of the publication.

subscription-value A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.

subscriber-id The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be **started** when publication updates are being sent from the consolidated database to the remote database.

The STOP SUBSCRIPTION statement prevents any further messages being sent to the subscriber. The START SUBSCRIPTION statement is required to restart messages being sent to the subscriber. However, you should ensure that the subscription is properly synchronized before restarting: that no messages have been missed.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "DROP SUBSCRIPTION statement [SQL Remote]" on page 509
- "START SUBSCRIPTION statement [SQL Remote]" on page 679
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 689
- "Database Extraction utility" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement stops the subscription of user **SamS** to the **pub_contact** publication.

```
STOP SUBSCRIPTION TO pub_contact
FOR SamS;
```


STOP SYNCHRONIZATION DELETE statement [MobiLink]

Use this statement to temporarily stop logging of deletes for MobiLink synchronization.

Syntax

STOP SYNCHRONIZATION DELETE

Remarks

Ordinarily, SQL Anywhere and UltraLite remote databases automatically log any changes made to tables or columns that are included in a synchronization, and then upload these changes to the consolidated database during the next synchronization. This statement allows you to temporarily suspend logging of delete operations to a SQL Anywhere or UltraLite remote database.

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the subsequent delete operations executed on that connection are synchronized. The effect continues until a START SYNCHRONIZATION DELETE statement is executed.

Repeating STOP SYNCHRONIZATION DELETE has no additional effect. A single START SYNCHRONIZATION DELETE statement restarts the logging, regardless of the number of STOP SYNCHRONIZATION DELETE statements preceding it.

This command can be useful to make corrections to a remote database, but should be used with caution as it effectively disables MobiLink synchronization.

Do not use STOP SYNCHRONIZATION DELETE if your application does not synchronize data.

Permissions

Must have DBA authority.

Side Effects

None.

See also

- UltraLite "StartSynchronizationDelete method" [*UltraLite - .NET Programming*]
- UltraLite "StopSynchronizationDelete method" [*UltraLite - .NET Programming*]
- "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 681

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

For an example, see "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 681.

SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]

Use this statement to synchronize a subscription for a user to a publication.

Syntax

SYNCHRONIZE SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *remote-user*, …

Parameters

publication-name The name of the publication to which the user is being subscribed. This may include the owner of the publication.

subscription-value A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.

remote-user The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be **synchronized** when the data in the remote database is consistent with that in the consolidated database, so that publication updates sent from the consolidated database to the remote database will not result in conflicts and errors.

To synchronize a subscription, a copy of the data in the publication at the consolidated database is sent to the remote database. The SYNCHRONIZE SUBSCRIPTION statement does this through the message system. It is recommended that where possible you use the database extraction utility (dbxtract) instead to synchronize subscriptions without using a message system.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 443
- "START SUBSCRIPTION statement [SQL Remote]" on page 679
- "STOP SUBSCRIPTION statement [SQL Remote]" on page 687
- "Database Extraction utility" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement synchronizes the subscription of user **SamS** to the **pub_contact** publication.


```
SYNCHRONIZE SUBSCRIPTION
   TO pub_contact
   FOR SamS;
```


SYSTEM statement [Interactive SQL]

Use this statement to launch an executable file from within Interactive SQL.

Syntax

SYSTEM ' [*path*] *file-name* '

Remarks

Launches the specified executable file.

- The SYSTEM statement must be entirely contained on one line.
- Comments are not allowed at the end of a SYSTEM statement.
- Enclose the path and file name in single quotation marks.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 370

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement launches the Notepad program, assuming that the Notepad executable is in your path.

```
SYSTEM 'notepad.exe';
```


TRIGGER EVENT statement

Use this statement to trigger a named event. The event may be defined for event triggers or be a scheduled event.

Syntax

TRIGGER EVENT *event-name* [**(** *parm* = *value*, … **)**]

Parameters

parm = value When a triggering condition causes an event handler to execute, the database server can provide context information to the event handler using the event_parameter function. The TRIGGER EVENT statement allows you to explicitly supply these parameters, to simulate a context for the event handler.

Remarks

Actions are tied to particular trigger conditions or schedules by a CREATE EVENT statement. You can use the TRIGGER EVENT statement to force the event handler to execute, even when the scheduled time or trigger condition has not occurred. TRIGGER EVENT does not execute disabled event handlers.

Permissions

Must have DBA authority.

Side effects

None.

See also

- “ALTER EVENT statement” on page 308
- “CREATE EVENT statement” on page 390
- “EVENT_PARAMETER function [System]” on page 160

Example

The following example shows how to pass a string parameter to an event. The event displays the time it was triggered on the database server console.

```
CREATE EVENT ev_PassedParameter
HANDLER
BEGIN
  MESSAGE 'ev_PassedParameter - was triggered at ' || event_parameter
( 'time' );
END;
TRIGGER EVENT ev_PassedParameter( "Time"=string( current timestamp ) );
```


TRUNCATE TABLE statement

Use this statement to delete all rows from a table, without deleting the table definition.

Syntax

TRUNCATE TABLE [*owner*.]*table-name*

Remarks

The TRUNCATE TABLE statement deletes all rows from a table. It is equivalent to a DELETE statement without a WHERE clause, except that no triggers are fired as a result of the TRUNCATE TABLE statement and each individual row deletion is not entered into the transaction log.

Note
This statement should be used with great care on a database involved in synchronization or replication. The TRUNCATE TABLE statement deletes all rows from a table, similar to a DELETE statement that doesn't have a WHERE clause. However, no triggers are fired as a result of a TRUNCATE TABLE statement. Furthermore, the row deletions are not entered into the transaction log and therefore are not synchronized or replicated. This can lead to inconsistencies that can cause synchronization or replication to fail.

After a TRUNCATE TABLE statement, the table structure and all of the indexes continue to exist until you issue a DROP TABLE statement. The column definitions and constraints remain intact, and triggers and permissions remain in effect.

Note
The TRUNCATE TABLE statement is entered into the transaction log as a single statement, like data definition statements. Each deleted row is not entered into the transaction log.
TRUNCATE TABLE statements are not replicated or synchronized by dbmlsync, dbremote, and dbltm. Only DELETE statements are replicated or synchronized.

If the truncate_with_auto_commit option is set to On (the default), and all the following criteria are satisfied, a fast form of table truncation is executed:

- There are no foreign keys either to or from the table.
- The TRUNCATE TABLE statement is not executed within a trigger.
- The TRUNCATE TABLE statement is not executed within an atomic statement.

If a fast truncation is carried out, then a COMMIT is carried out before and after the operation. When a fast truncation is carried out, individual DELETEs are not recorded in the transaction log.

Permissions

Must be the table owner, or have DBA authority, or have ALTER permissions on the table.

For base tables, the TRUNCATE TABLE statement requires exclusive access to the table, as the operation is atomic (either all rows are deleted, or none are). This means that any cursors that were previously opened

and that reference the table being truncated must be closed and a COMMIT or ROLLBACK must be issued to release the reference to the table.

For temporary tables, each user has their own copy of the data, and exclusive access is not required.

If a fast truncation is performed, TRUNCATE TABLE cannot be used within snapshot transactions. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Delete triggers are not fired by the TRUNCATE TABLE statement.

If truncate_with_auto_commit is set to On, then a COMMIT is performed before and after the table is truncated.

Individual deletions of rows are not entered into the transaction log, so the TRUNCATE TABLE operation is not replicated. Do not use this statement in SQL Remote replication or on a MobiLink remote database.

If the table contains a column defined as DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT, TRUNCATE TABLE resets the next available value for the column.

See also

- "DELETE statement" on page 485
- "truncate_with_auto_commit option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

Delete all rows from the Departments table.

```
TRUNCATE TABLE Departments;
```


UNION statement

Use this statement to combine the results of two or more select statements.

Syntax

[**WITH** *temporary-views*] *query-block*
UNION [**ALL** | **DISTINCT**] *query-block*
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], …]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

Note
You cannot use the FOR, FOR XML, WITH, or OPTION clause in the *query-block*.

OPTION clause

This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].

- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The results of several query blocks can be combined into a larger result using UNION. Each *query-block* must have the same number of items in the select list.

The results of UNION ALL are the combined results of the query blocks. The results of UNION are the same as UNION ALL, except that duplicate rows are eliminated. Eliminating duplicates requires extra processing, so UNION ALL should be used instead of UNION where possible. UNION DISTINCT is identical to UNION.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately.

The first query block of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None.

See also

- "SELECT statement" on page 648

Standards and compatibility

- **SQL/2003** Core feature.

Example

List all distinct surnames of employees and customers.

```
SELECT Surname
FROM Employees
UNION
```



```
SELECT Surname
FROM Customers;
```


UNLOAD statement

Use this statement to export data from a database into an external ASCII-format file.

Syntax

UNLOAD *select-statement*
TO *file-name*
[*unload-option* …]

unload-option :
APPEND {**ON**|**OFF**}
| **DELIMITED BY** *string*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** {**ON** | **OFF**}
| **FORMAT** {**ASCII** | **BCP**}
| **HEXADECIMAL** {**ON** | **OFF**}
| **QUOTE** *string*
| **QUOTES** {**ON** | **OFF**}

file-name : *string* | *variable*

Parameters

file-name The file name to which the data is to be unloaded. Because it is the database server that executes the statements, *file-name* specifies a file on the database server computer. A relative *file-name* specifies a file relative to the database server's starting directory. To unload data onto a client computer, see "PASSTHROUGH statement [SQL Remote]" on page 609.

Remarks

The UNLOAD statement allows the result set of a query to be exported to a comma-delimited file. The result set is not ordered unless the query itself contains an ORDER BY clause.

When unloading result set columns with binary data types, UNLOAD writes hexadecimal strings, of the form *\0xnnnn* where *n* is a hexadecimal digit.

For a description of the *unload-option* parameters, see "UNLOAD TABLE statement" on page 700.

When unloading and reloading a database that has proxy tables, you must create an external login to map the local user to the remote user, even if the user has the same password on both the local and remote databases. If you do not have an external login, the reload may fail because you cannot connect to the remote server.

For more information about external logins, see "Working with external logins" [*SQL Anywhere Server - SQL Usage*].

When the APPEND option is ON, unloaded data is appended to the end of the file specified. When the APPEND option is OFF, unloaded data replaces the contents of the file specified. This option is OFF by default.

Permissions

The permissions required to execute an UNLOAD statement are set on the database server command line, using the –gl option. See "-gl server option" [*SQL Anywhere Server - Database Administration*].

Side effects

None. The query is executed at the current isolation level.

See also

- "UNLOAD TABLE statement" on page 700
- "OUTPUT statement [Interactive SQL]" on page 604

Standards and compatibility

- **SQL/2003** Vendor extension.

UNLOAD TABLE statement

Use this statement to export data from a database table, or from a materialized view, into an external file.

Syntax

UNLOAD [**FROM**] {
[**TABLE**] [*owner.*]*table-name*
| [**MATERIALIZED VIEW**] [*owner.*]*materialized-view-name* }
TO *file-name*
[*unload-option* ...]

unload-option :
APPEND { **ON** | **OFF** }
| **DELIMITED BY** *string*
| **ENCODING** *encoding*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **ASCII** | **BCP** }
| **HEXADECIMAL** { **ON** | **OFF** }
| **ORDER** { **ON** | **OFF** }
| **QUOTE** *string*
| **QUOTES** { **ON** | **OFF** }
| **ROW DELIMITED BY** *string*

file-name : { *string* | *variable* }

encoding : *string*

Parameters

file-name The file to which the data is to be unloaded. Because the database server executes the statements, file names specify files on the database server computer. Relative file names specify files relative to the database server's starting directory. To unload data onto a client computer, see "PASSTHROUGH statement [SQL Remote]" on page 609.

APPEND option When the APPEND option is ON, unloaded data is appended to the end of the file specified. When the APPEND option is OFF, unloaded data replaces the contents of the file specified. This option is OFF by default.

DELIMITED BY The string used between columns. The default column delimiter is a comma. You can specify an alternative column delimiter by providing a string. However, only the first byte (character) of the string is used as the delimiter.

ENCODING option All database data is translated from the database character encoding to the specified character encoding. When ENCODING is not specified, the database's character encoding is used, and translation is not performed.

For more information on how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

If a translation error occurs during the unload operation, it is reported based on the setting of the on_charset_conversion_failure option. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

The following example unloads the data using the UTF-8 character encoding:

```
UNLOAD TABLE mytable TO 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
```

ESCAPES option With ESCAPES on (the default), backslash-character combinations are used to identify special characters where necessary on export.

FORMAT option Outputs data in either ASCII format or in BCP out format.

HEXADECIMAL option By default, HEXADECIMAL is ON. Binary column values are written as **0x***nnnnnn*…, where each *n* is a hexadecimal digit. It is important to use HEXADECIMAL ON when dealing with multibyte character sets.

The HEXADECIMAL option can be used only with the FORMAT ASCII option.

ORDER option With ORDER ON (the default), the exported data is ordered by clustered index if one exists. If a clustered index does not exist, the exported data is ordered by primary key values. With ORDER OFF, the data is exported in the same order you see when selecting from the table without an ORDER BY clause.

Exporting is slower with ORDER ON. However, reloading using the LOAD TABLE statement is quicker because of the simplicity of the indexing step.

For more information on clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

QUOTE option The QUOTE clause is for ASCII data only; the *string* is placed around string values. The default is a single quote (apostrophe).

QUOTES option With QUOTES turned on (the default), single quotes are placed around all exported strings.

ROW DELIMITED BY option Use this clause to specify the string that indicates the end of a record. The default delimiter string is a comma. You can specify an alternative delimiter by providing a string up to 255 bytes in length; for example, `... ROW DELIMITED BY '###' ...`. The same formatting requirements apply as to other SQL strings. If you wanted to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `... ROW DELIMITED BY '\x09' ...`. If your delimiter string contains a \n, it will match either \r\n or \n.

Remarks

The UNLOAD TABLE statement allows efficient mass exporting from a database table or materialized view into a file. The UNLOAD TABLE statement is more efficient than the Interactive SQL statement OUTPUT, and can be called from any client application.

UNLOAD TABLE places an exclusive lock on the whole table.

When unloading columns with binary data types, UNLOAD TABLE writes hexadecimal strings, of the form \x*nnnn*, where *n* is a hexadecimal digit.

For descriptions of the FORMAT and ESCAPE CHARACTER options, see "LOAD TABLE statement" on page 585.

Permissions

The permissions required to execute an UNLOAD TABLE statement depend on the database server –gl option, as follows:

- If the -gl option is ALL, you must have SELECT permissions on the table or tables referenced in the UNLOAD TABLE statement.
- If the -gl option is DBA, you must have DBA authority.
- If the -gl option is NONE, UNLOAD TABLE is not permitted.

See "-gl server option" [*SQL Anywhere Server - Database Administration*].

Side effects

None.

See also

- "LOAD TABLE statement" on page 585
- "OUTPUT statement [Interactive SQL]" on page 604
- "UNLOAD statement" on page 698

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example unloads UTF-8-encoded table data into mytable:

```
LOAD TABLE mytable TO 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
```


UPDATE statement

Use this statement to modify existing rows in database tables.

Syntax 1

UPDATE [**FIRST** | **TOP** *n*] *table-list*
SET *set-item*, …
[**FROM** *table-list*]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], …]
[**OPTION(** *query-hint*, ... **)**]

Syntax 2

UPDATE *table-name*
SET *set-item*, …
VERIFY (*column-name*, … **) VALUES (** *expression*, … **)**
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], …]
[**OPTION(** *query-hint*, ... **)**]

Syntax 3

UPDATE *table*
PUBLICATION *publication*
{ **SUBSCRIBE BY** *expression*
| **OLD SUBSCRIBE BY** *expression* **NEW SUBSCRIBE BY** *expression*
}
WHERE *search-condition*

set-item :
column-name [.*field-name*…] **=** *expression*
| *@variable-name* **=** *expression*

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

UPDATE clause The table is either a base table, a temporary table, or a view. Views can be updated unless the SELECT statement defining the view contains a GROUP BY clause or aggregate function, or involves a UNION statement.

FIRST or TOP clause Primarily for use with the ORDER BY clause, this clause allows you to update only a certain subset of the rows that satisfy the WHERE clause. You cannot use a variable as input with FIRST or TOP.

SET clause The set clause specifies the columns and how the values are changed.

You can use the SET clause to set the column to a computed column value using this format:


```
SET column-name = expression, ...
```

Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the *expression*. If the expression is a *column-name*, the old value is used.

You can also use the SET clause to assign a variable using this format:

```
SET @variable-name = expression, ...
```

When assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table as well as the variable name, the statement will update the column.

Following is an example of part of an UPDATE statement. It assigns a variable in addition to updating the table:

```
UPDATE T SET @var = expression1, col1 = expression2
WHERE...
```

This is equivalent to:

```
SELECT @var = expression1
FROM T
WHERE... ;
UPDATE T SET col1 = expression2
WHERE...
```

FROM clause The optional FROM clause allows tables to be updated based on joins. If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause. Data is updated only in the table list of the UPDATE clause.

If a FROM clause is used, it is important to qualify the table name the same way in both parts of the statement. If a correlation name is used in one place, the same correlation name must be used elsewhere. Otherwise, an error is generated.

This clause is allowed only if ansi_update_constraints is set to Off. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

For a full description of joins, see "Joins: Retrieving Data from Several Tables" [*SQL Anywhere Server - SQL Usage*].

For more information, see "FROM clause" on page 535.

WHERE clause If a WHERE clause is specified, only rows satisfying the search condition are updated. If no WHERE clause is specified, every row is updated.

ORDER BY clause Normally, the order in which rows are updated does not matter. However, in conjunction with the FIRST or TOP clause the order can be significant.

You cannot use ordinal column numbers in the ORDER BY clause.

You must not update columns that appear in the ORDER BY clause unless you set the ansi_update_constraints option to Off. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

OPTION clause

This clause provides hints as to how to process the query. The following query hints are supported:

- **MATERIALIZED VIEW OPTIMIZATION '*option-value*'** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*].
- **FORCE OPTIMIZATION** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

  Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

  For more information on simple queries and view matching, see "Phases of query processing" [*SQL Anywhere Server - SQL Usage*], and "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].
- ***option-name = option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:
  - "isolation_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]

Case sensitivity Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. A CHAR data type column updated with a string Value is always held in the database with an uppercase V and the remainder of the letters lowercase. SELECT statements return the string as Value. If the database is not case sensitive, however, all comparisons make Value the same as value, VALUE, and so on. Further, if a single-column primary key already contains an entry Value, an INSERT of value is rejected, as it would make the primary key not unique.

Updates that leave a row unchanged If the new value does not differ from the old value, no change is made to the data. However, BEFORE UPDATE triggers fire any time an UPDATE occurs on a row, whether or not the new value differs from the old value. AFTER UPDATE triggers fire only if the new value is different from the old value.

Remarks

Syntax 1 of the UPDATE statement modifies values in rows of one or more tables. Syntax 2 and 3 are applicable only to SQL Remote.

Syntax 2 is intended for use with SQL Remote only, in single-row updates of a single table executed by the Message Agent. The VERIFY clause contains a set of values that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail simply because the VERIFY clause fails to match.

Syntax 3 of the UPDATE statement is used to implement a specific SQL Remote feature, and is to be used inside a BEFORE trigger. It provides a full list of SUBSCRIBE BY values any time the list changes. It is placed in SQL Remote triggers so that the database server can compute the current list of SUBSCRIBE BY values. Both lists are placed in the transaction log.

The Message Agent uses the two lists to make sure that the row moves to any remote database that did not have the row and now needs it. The Message Agent also removes the row from any remote database that has the row and no longer needs it. A remote database that has the row and still needs it is not be affected by the UPDATE statement.

For publications created using a subquery in a SUBSCRIBE BY clause, you must write a trigger containing syntax 3 of the UPDATE statement to ensure that the rows are kept in their proper subscriptions.

Syntax 3 of the UPDATE statement allows the old SUBSCRIBE BY list and the new SUBSCRIBE BY list to be explicitly specified, which can make SQL Remote triggers more efficient. In the absence of these lists, the database server computes the old SUBSCRIBE BY list from the publication definition. Since the new SUBSCRIBE BY list is commonly only slightly different from the old SUBSCRIBE BY list, the work to compute the old list may be done twice. By specifying both the old and new lists, you can avoid this extra work.

The SUBSCRIBE BY expression is either a value or a subquery.

Syntax 3 of the UPDATE statement makes an entry in the transaction log, but does not change the database table.

Updating a significant amount of data using the UPDATE statement also updates column statistics.

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

Column statistics are updated.

See also

- “DELETE statement” on page 485
- “INSERT statement” on page 573
- “FROM clause” on page 535
- “Joins: Retrieving Data from Several Tables” [*SQL Anywhere Server - SQL Usage*]
- “UPDATE (positioned) statement [ESQL] [SP]” on page 708

Standards and compatibility

- **SQL/2003** Syntax 1 is a core feature, except for the FROM and ORDER BY clauses, which are vendor extensions. Syntax 2 and 3 are vendor extensions for use only with SQL Remote.

To enforce SQL/2003 compatibility, ensure that the ansi_update_constraints option is set to Strict. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Example

Using the sample database, this example transfers employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE Employees
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

Using the sample database, this example renumbers all existing sales orders by subtracting 2000 from the ID.

```
UPDATE SalesOrders AS orders
SET orders.ID = orders.ID - 2000
ORDER BY orders.ID ASC;
```

This update is possible only if the foreign key of the SalesOrderItems table (referencing the primary key SalesOrders.ID) is defined with the action ON UPDATE CASCADE. The SalesOrderItems table is then updated as well.

For more information on foreign key properties, see "ALTER TABLE statement" on page 332 and "CREATE TABLE statement" on page 450.

Using the sample database, this example changes the price of a product at isolation level 2, rather than using the current isolation level setting of the database.

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501
OPTION( isolation_level = 2 );
```


UPDATE (positioned) statement [ESQL] [SP]

Use this statement to modify the data at the current location of a cursor.

Syntax 1

UPDATE WHERE CURRENT OF *cursor-name*
{ **USING DESCRIPTOR** *sqlda-name* | **FROM** *hostvar-list* }

Syntax 2

UPDATE *update-table*, ...
SET *set-item*, ...
WHERE CURRENT OF *cursor-name*

hostvar-list : *indicator variables allowed*

update-table :
[*owner-name.*]*table-or-view-name* [[**AS**] *correlation-name*]

set-item :
[*correlation-name.*]*column-name* **=** *expression*
| [*owner-name.*]*table-or-view-name***.***column-name* **=** *expression*

sqlda-name : *identifier*

Parameters

USING DESCRIPTOR clause When assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table as well as the variable name, the statement will update the column.

SET clause The columns that are referenced in *set-item* must be in the table or view that is updated. They cannot refer to aliases, nor to columns from other tables or views. If the table or view you are updating is given a correlation name in the cursor specification, you must use the correlation name in the SET clause.

Each *set-item* is associated with a single *update-table*, and the corresponding column of the matching table in the cursor's query is modified. The *expression* references columns of the tables identified in the UPDATE list and may use constants, variables, expressions from the select list of the query, or combinations of the above using operators such as +, -, …, COALESCE, IF, and so on. The *expression* can not reference aliases of expressions from the cursor's query, nor can they reference columns of other tables of the cursor's query which do not appear in the UPDATE list. Subselects, subquery predicates, and aggregate functions can not be used in the *set-items*.

Each *update-table* is matched to a table in the query for the cursor as follows:

- If a correlation name is specified, it is matched to a table in the cursor's query that has the same *table-or-view-name* and the same *correlation-name*.
- Otherwise, if there is a table in the cursor's query that has the same *table-or-view-name* that does not have a correlation name specified, or has a correlation name that is the same as the *table-or-view-name*, then the update table is matched with this table in the cursor's query.

- Otherwise, if there is a single table in the cursor's query that has the same *table-or-view-name* as the update table, then the update table is matched with this table in the cursor's query.

Remarks

This form of the UPDATE statement updates the current row of the specified cursor. The current row is defined to be the last row successfully fetched from the cursor, and the last operation on the cursor must not have been a positioned DELETE statement.

For syntax 1, columns from the SQLDA or values from the host variable list correspond one-to-one with the columns returned from the specified cursor. If the sqldata pointer in the SQLDA is the null pointer, the corresponding select list item is not updated.

In syntax 2, the requested columns are set to the specified values for the row at the current row of the specified query. The columns do not need to be in the select list of the specified open cursor. This format can be prepared.

Also, when assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table as well as the variable name, the statement will update the column.

The USING DESCRIPTOR, FROM *hostvar-list*, and *hostvar* formats are for embedded SQL only.

Permissions

Must have UPDATE permission on the columns being modified.

Side effects

None.

See also

- “DELETE statement” on page 485
- “DELETE (positioned) statement [ESQL] [SP]” on page 488
- “UPDATE statement” on page 703

Standards and compatibility

- **SQL/2003** Core feature. The range of cursors that can be updated may contain vendor extensions if the ansi_update_constraints option is set to Off.
- **Sybase** Embedded SQL use is supported by Open Client/Open Server, and procedure and trigger use is supported in SQL Anywhere.

Example

The following is an example of an UPDATE statement WHERE CURRENT OF cursor:

```
UPDATE Employees
SET Surname = 'Jones'
WHERE CURRENT OF emp_cursor;
```


UPDATE statement [SQL Remote]

Use this statement to modify data in the database.

Syntax 1

UPDATE *table-list*
SET *column-name* = *expression*, …
[**VERIFY (** *column-name*, … **) VALUES (** *expression*, … **)**]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], …]

Syntax 2

UPDATE *table*
PUBLICATION *publication*
{ **SUBSCRIBE BY** *expression* |
  OLD SUBSCRIBE BY *expression*
  NEW SUBSCRIBE BY *expression* }
WHERE *search-condition*

expression: *value* | *subquery*

Remarks

Syntax 1 and Syntax 2 are applicable only to SQL Remote.

Syntax 2 with no OLD and NEW SUBSCRIBE BY expressions must be used in a BEFORE trigger.

Syntax 2 with OLD and NEW SUBSCRIBE BY expressions can be used anywhere.

The UPDATE statement is used to modify rows of one or more tables. Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the *expression*. Even *column-name* can be used in the expression—the old value is used.

If no WHERE clause is specified, every row is updated. If a WHERE clause is specified, then only those rows which satisfy the search condition are updated.

Normally, the order that rows are updated does not matter. However, in conjunction with the NUMBER(*) function, an ordering can be useful to get increasing numbers added to the rows in some specified order. Also, if you want to do something like add 1 to the primary key values of a table, it is necessary to do this in descending order by primary key, so that you do not get duplicate primary keys during the operation.

Views can be updated provided the SELECT statement defining the view does not contain a GROUP BY clause, an aggregate function, or involve a UNION statement.

Character strings inserted into tables are always stored in the case they are entered, regardless of whether the database is case sensitive or not. Thus a character data type column updated with a string Value is always held in the database with an upper-case V and the remainder of the letters lowercase. SELECT statements return the string as Value. If the database is not case sensitive, however, all comparisons make Value the same as value, VALUE, and so on. Further, if a single-column primary key already contains an entry Value, an INSERT of value is rejected, as it would make the primary key not unique.

The optional FROM clause allows tables to be updated based on joins. If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause. Data is updated only in the table list immediately following the UPDATE keyword.

If a FROM clause is used, it is important to qualify the table name that is being updated the same way in both parts of the statement. If a correlation name is used in one place, the same correlation name must be used in the other. Otherwise, an error is generated.

Syntax 1 is intended for use with SQL Remote only, in single-row updates executed by the Message Agent. The VERIFY clause contains a set of values that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail if the VERIFY clause fails to match. When the VERIFY clause is specified, only one table can be updated at a time.

Syntax 2 is intended for use with SQL Remote only. If no OLD and NEW expressions are used, it must be used inside a BEFORE trigger so that it has access to the relevant values. The purpose is to provide a full list of subscribe by values any time the list changes. It is placed in SQL Remote triggers so that the database server can compute the current list of SUBSCRIBE BY values. Both lists are placed in the transaction log.

The Message Agent uses the two lists to make sure that the row moves to any remote database that did not have the row and now needs it. The Message Agent also removes the row from any remote database that has the row and no longer needs it. A remote database that has the row and still needs it is not be affected by the UPDATE statement.

Syntax 2 of the UPDATE statement allows the old SUBSCRIBE BY list and the new SUBSCRIBE BY list to be explicitly specified, which can make SQL Remote triggers more efficient. In the absence of these lists, the database server computes the old SUBSCRIBE BY list from the publication definition. Since the new SUBSCRIBE BY list is commonly only slightly different from the old SUBSCRIBE BY list, the work to compute the old list may be done twice. By specifying both the old and new lists, this extra work can be avoided.

The OLD and NEW SUBSCRIBE BY syntax is especially useful when many tables are being updated in the same trigger with the same subscribe by expressions. This can dramatically increase performance.

The SUBSCRIBE BY expression is either a value or a subquery.

Syntax 2 of the UPDATE statement is used to implement a specific SQL Remote feature, and is to be used inside a BEFORE trigger.

For publications created using a subquery in a subscription expression, you must write a trigger containing syntax 2 of the UPDATE statement to ensure that the rows are kept in their proper subscriptions.

For a full description of this feature, see “Territory realignment in the Contacts example” [*SQL Remote*].

Syntax 2 of the UPDATE statement makes an entry in the transaction log, but does not change the database table.

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

None.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example transfers employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE Employees
VERIFY( DepartmentID ) VALUES( 300 )
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

VALIDATE statement

Use this statement to validate the current database, or a table or materialized view in the current database.

Syntax 1 - Validating tables and materialized views

VALIDATE {
TABLE [*owner*.]*table-name*
| **MATERIALIZED VIEW** [*owner*.]*materialized-view-name* }
[**WITH EXPRESS CHECK**]

Syntax 2 - Validating a database

VALIDATE { **CHECKSUM** | **DATABASE** }

Syntax 3 - Validating indexes

VALIDATE {
INDEX *index-name*
| [**INDEX**] **FOREIGN KEY** *role-name*
| [**INDEX**] **PRIMARY KEY** }
ON [*owner*.]*object-name*
}

object-name : *table-name* | *materialized-view-name*

Parameters

WITH EXPRESS CHECK In addition to the default checks, check that the number of rows in the table or materialized view matches the number of entries in the index. This option does not perform individual index lookups for each row, nor does it perform checksum validation. This option can significantly improve performance when validating large databases with a small cache.

Remarks

Validation of tables includes a checksum validation, and validation that the number of rows in a table matches the number of rows in each index associated with the table. If you specify WITH EXPRESS CHECK, a checksum validation is not performed.

The VALIDATE DATABASE statement validates that all table pages in the database belong to the correct object. VALIDATE DATABASE also performs a checksum validation, but does not validate the indexes.

Use the VALIDATE CHECKSUM statement to perform a checksum validation on the database. The VALIDATE CHECKSUM statement ensures that database pages have not been modified on disk. When a database is created with checksums enabled, a checksum is calculated for each database page before it is written to disk. VALIDATE CHECKSUM reads each database page from disk and calculates the checksum for each page. If the calculated checksum for a page does not match the stored checksum for that page, an error occurs and information about the invalid page appears in the Server Messages window. The VALIDATE CHECKSUM statement can also be useful on databases with checksums disabled, since critical database pages still include checksums.

Use the VALIDATE INDEX statement to validate an index, including index statistics, on a table or a materialized view. The VALIDATE INDEX statement ensures that every row referenced in the index actually exists. For foreign key indexes, it also ensures that the corresponding row exists in the primary table. This check complements the validity checking carried out by the VALIDATE TABLE statement. The

VALIDATE INDEX statement also verifies that the statistics reported on the specified indexes are accurate. If they are not accurate, an error is generated.

Caution
Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, spurious errors may be reported indicating some form of database corruption even though no corruption actually exists.

Permissions

Must have DBA or VALIDATE authority.

Side effects

None.

See also

- "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*]
- "sa_validate system procedure" on page 934
- "Ensuring your database is valid" [*SQL Anywhere Server - Database Administration*]
- "CREATE DATABASE statement" on page 374
- "CREATE INDEX statement" on page 405

Standards and compatibility

- **SQL/2003** Vendor extension.

WAITFOR statement

Use this statement to delay processing for the current connection for a specified amount of time or until a given time.

Syntax

WAITFOR { **DELAY** *time* | **TIME** *time* }
[**CHECK EVERY** *integer*]
[**AFTER MESSAGE BREAK**]

time : *string*

Parameters

DELAY If DELAY is used, processing is suspended for the given interval.

TIME If TIME is specified, processing is suspended until the database server time reaches the time specified.

If the current server time is greater than the time specified, processing is suspended until that time on the following day.

CHECK EVERY This optional clause controls how often the WAITFOR statement wakes up. By default, it wakes up every 5 seconds. The value is in milliseconds, and the minimum value is 250 milliseconds.

AFTER MESSAGE BREAK The WAITFOR statement can be used to wait for a message from another connection. In most cases, when a message is received it is forwarded to the application that executed the WAITFOR statement and the WAITFOR statement continues to wait. If the AFTER MESSAGE BREAK clause is specified, when a message is received from another connection, the WAITFOR statement completes. The message text is not forwarded to the application, but it can be accessed by obtaining the value of the MessageReceived connection property.

For more information about the MessageReceived property, see "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

Remarks

The WAITFOR statement wakes up periodically (every 5 seconds by default) to check if it has been canceled or if messages have been received. If neither of these has happened, the statement continues to wait.

WAITFOR provides an alternative to the following statement:

```
CALL java.lang.Thread.sleep( <time_to_wait_in_millisecs> );
```

In many cases, scheduled events are a better choice than using WAITFOR TIME, because scheduled events execute on their own connection.

Permissions

None

Side effects

The implementation of this statement uses a worker thread while it is waiting. This uses up one of the threads specified by the -gn database option (the default is 20 threads).

See also

- "CREATE EVENT statement" on page 390

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example waits for three seconds:

```
WAITFOR DELAY '00:00:03';
```

The following example waits for 0.5 seconds (500 milliseconds):

```
WAITFOR DELAY '00:00:00:500';
```

The following example waits until 8 PM:

```
WAITFOR TIME '20:00';
```

In the following example, connection 1's WAITFOR statement completes when it receives the message from connection 2:

```
// connection 1:
BEGIN
  DECLARE msg LONG VARCHAR;
  LOOP  // forever
    WAITFOR DELAY '00:05:00' AFTER MESSAGE BREAK;
    SET msg = CONNECTION_PROPERTY('MessageReceived');
    IF msg != '' THEN
      MESSAGE 'Msg: ' || msg TO CONSOLE;
    END IF;
  END LOOP
END
// connection 2:
MESSAGE 'here it is' FOR connection 1
```


WHENEVER statement [ESQL]

Use this statement to specify error handling in embedded SQL programs.

Syntax

WHENEVER { **SQLERROR** | **SQLWARNING** | **NOTFOUND** }
GOTO *label* | **STOP** | **CONTINUE** | { *C-code*; }

label : *identifier*

Remarks

The WHENEVER statement is used to trap errors, warnings and exceptional conditions encountered by the database when processing SQL statements. The statement can be put anywhere in an embedded SQL program and does not generate any code. The preprocessor will generate code following each successive SQL statement. The error action remains in effect for all embedded SQL statements from the source line of the WHENEVER statement until the next WHENEVER statement with the same error condition, or the end of the source file.

Errors based on source position
The error conditions are in effect based on positioning in the C language source file, not based on when the statements are executed.

The default action is CONTINUE.

Note that this statement is provided for convenience in simple programs. Most of the time, checking the sqlcode field of the SQLCA (SQLCODE) directly is the easiest way to check error conditions. In this case, the WHENEVER statement would not be used. If fact, all the WHENEVER statement does is cause the preprocessor to generate an *if (SQLCODE)* test after each statement.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following are examples of the WHENEVER statement:

```
EXEC SQL WHENEVER NOTFOUND GOTO done;
EXEC SQL WHENEVER SQLERROR
   {
      PrintError( &sqlca );
      return( FALSE );
   };
```


WHILE statement [T-SQL]

Use this statement to provide repeated execution of a statement or compound statement.

Syntax

WHILE *search-condition-statement*

Remarks

The WHILE conditional affects the execution of only a single SQL statement, unless statements are grouped into a compound statement between the keywords BEGIN and END.

The BREAK statement and CONTINUE statement can be used to control execution of the statements in the compound statement. The BREAK statement terminates the loop, and execution resumes after the END keyword marking the end of the loop. The CONTINUE statement causes the WHILE loop to restart, skipping any statements after the CONTINUE.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 595

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following code illustrates the use of WHILE:

```
WHILE ( SELECT AVG(UnitPrice) FROM Products ) < $30
BEGIN
   UPDATE Products
   SET UnitPrice = UnitPrice + 2
   IF ( SELECT MAX(UnitPrice) FROM Products ) > $50
      BREAK
END
```

The BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise, the loop continues until the average price is greater than or equal to $30.

WINDOW clause

Use the WINDOW clause in a SELECT statement to define all or part of a window for use with window functions such as AVG and RANK.

Syntax

WINDOW *window-expression*, ...

window-expression : *new-window-name* **AS (** *window-spec* **)**

window-spec :
[*existing-window-name*]
[**PARTITION BY** *expression*, ...]
[**ORDER BY** *expression* [**ASC** | **DESC**], ...]
[{ **ROWS** | **RANGE** } { *window-frame-start* | *window-frame-between* }]

window-frame-start :
{ **UNBOUNDED PRECEDING**
| *unsigned-integer* **PRECEDING**
| **CURRENT ROW** }

window-frame-between :
BETWEEN *window-frame-bound1* **AND** *window-frame-bound2*

window-frame-bound :
window-frame-start
| **UNBOUNDED FOLLOWING**
| *unsigned-integer* **FOLLOWING**

Parameters

PARTITION BY clause The PARTITION BY clause organizes the result set into logical groups based on the unique values of the specified expression. When this clause is used with window functions, the functions are applied to each partition independently. For example, if you follow PARTITION BY with a column name, the result set is partitioned by distinct values in the column.

If this clause is omitted, the entire result set is considered a partition.

ORDER BY clause The ORDER BY clause defines how to sort the rows in each partition of the result set. You can further control the order by specifying ASC for ascending order (the default) or DESC for descending order.

If this clause is omitted, SQL Anywhere returns rows in whatever order is most efficient. This means that the appearance of result sets may vary depending on when you last accessed the row, and other factors.

ROWS clause and RANGE clause Use either a ROWS or RANGE clause to express the size of the window. The window size can be one, many, or all rows of a partition. You can express the size of the window either in terms of a range of data values offset from the value in the current row (RANGE), or in terms of the number of rows offset from the current row (ROWS).

When using the RANGE clause, you must also use an ORDER BY clause. This is because the calculation performed to produce the window requires that the values be sorted. Additionally, the ORDER BY clause cannot contain more than one expression, and the expression must result in either a date or a numeric value.

When using the ROWS or RANGE clauses, if you specify only a starting row, the current row is used as the last row in the window. If you specify only an ending row, the current row is used as the first row.

- **PRECEDING clause** Use the PRECEDING clause to define the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. For example, `5 PRECEDING` sets the window to start with the fifth row preceding the current row.

  Use UNBOUNDED PRECEDING to set the first row in the window to be the first row in the partition.

- **BETWEEN clause** Use the BETWEEN clause to define the first and last row of the window, using the current row as a reference point. First and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. For example, `BETWEEN 3 PRECEDING AND 5 FOLLOWING` sets the window to start with the third row preceding the current row, and end with the fifth row following the current row.

  Use BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to set the first and last rows in the window to be the first and last row in the partition, respectively. This is equivalent to the default behavior if no ROW or RANGE clause is specified.

- **FOLLOWING clause** Use the FOLLOWING clause to define the last row of the window using the current row as a reference point. The last row is expressed in terms of the number of rows following the current row.

  Use UNBOUNDED FOLLOWING to set the last row in the window to be the last row in the partition.

If you do not specify a ROW or a RANGE clause, the window size is determined as follows:

- If an ORDER BY clause is specified, the window starts with the first row in the partition (UNBOUNDED PRECEDING) and ends with the current row (CURRENT ROW).
- If an ORDER BY clause is not specified, the window starts with the first row in the partition (UNBOUNDED PRECEDING) and ends with last row in the partition (UNBOUNDED FOLLOWING).

Remarks

The WINDOW clause must appear before the ORDER BY clause in a SELECT statement.

Depending on what you are trying to achieve with your results, you might specify all of the settings for a window in the WINDOW clause, and then name (refer to) the window from within the window function syntax (for example, `AVG() OVER` *window-name*). You could also specify the entire window in the window function and not use a WINDOW clause at all. Finally, you could also split the definition between the window function syntax, and the WINDOW clause. For example:

```
AVG() OVER ( windowA
             ORDER BY expression )...
...
WINDOW windowA AS ( PARTITION BY expression )
```


When splitting the window definition in this manner, the following restrictions apply:

- You cannot use a PARTITION BY clause in the window function syntax.
- You can use an ORDER BY clause in either the window function syntax or in the WINDOW clause, but not in both.
- You cannot include a RANGE or ROWS clause in the WINDOW clause.

With the exception of the LIST function, all aggregate functions can be used as window functions. However, ranking aggregate functions (RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST, and ROW_NUMBER) require an ORDER BY clause, and do not allow a ROW or RANGE clause in the WINDOW clause or inline definition. For all other window functions, you can use any of the clauses, depending on what you are trying to achieve.

For more information about how to define and use windows in order to achieve the results you want, see "Defining a window" [*SQL Anywhere Server - SQL Usage*].

See also

- "SELECT statement" on page 648
- "OLAP Support" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/2003 features T611, T612.

Example

The following example returns an employee's salary as well as the average salary for all employees in that State. The results are ordered by State and then by Surname.

```
SELECT EmployeeID, Surname, Salary, State,
  AVG( Salary ) OVER SalaryWindow
FROM Employees
WINDOW SalaryWindow AS ( PARTITION BY State )
ORDER BY State, Surname;
```


WRITETEXT statement [T-SQL]

Permits non-logged, interactive updating of an existing text or image column.

Syntax

WRITETEXT *table-name.column-name*
text-pointer [**WITH LOG**] *data*

Remarks

Updates an existing text or image value. The update is not recorded in the transaction log, unless the WITH LOG option is supplied. You cannot carry out WRITETEXT operations on views.

Permissions

None.

Side effects

WRITETEXT does not fire triggers, and by default WRITETEXT operations are not recorded in the transaction log.

See also

- "READTEXT statement [T-SQL]" on page 620
- "TEXTPTR function [Text and image]" on page 265

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following code fragment illustrates the use of the WRITETEXT statement. The SELECT statement in this example returns a single row. The example replaces the contents of the column_name column on the specified row with the value newdata.

```
EXEC SQL create variable textpointer binary(16);
EXEC SQL set textpointer =
   (  SELECT textptr(column_name)
      FROM table_name WHERE ID = 5 );
EXEC SQL writetext table_name.column_name
   textpointer 'newdata';
```


Part II. System Objects

This section describes SQL Anywhere tables, views, and procedures.

CHAPTER 5

Tables

Contents

System tables 726
Diagnostic tracing tables 735
Other tables 751

System tables

The structure of every database is described in a number of system tables. System tables are owned by the user SYS. The contents of these tables can be changed only by the database server. The UPDATE, DELETE, and INSERT commands cannot be used to modify the contents of these tables. Further, the structure of these tables cannot be changed using the ALTER TABLE and DROP commands.

System tables in SQL Anywhere are exposed via their corresponding views.

DUMMY system table

Column name	Column type	Column constraint	Table constraints
dummy_col	INTEGER	NOT NULL	

The DUMMY table is provided as a read-only table that always has exactly one row. This can be useful for extracting information from the database, as in the following example that gets the current user ID and the current date from the database.

```
SELECT USER, today(*) FROM SYS.DUMMY;
```

Use of SYS.DUMMY in the FROM clause is optional. If no table is specified in the FROM clause, the table is assumed to be SYS.DUMMY. The above example could be written as follows:

```
SELECT USER, today(*);
```

dummy_col This column is not used. It is present because a table cannot be created with no columns.

The cost of reading from the SYS.DUMMY table is less than the cost of reading from a similar user-created table because there is no latch placed on the table page of SYS.DUMMY.

Access plans are not constructed with scans of the SYS.DUMMY table. Instead, references to SYS.DUMMY are replaced with a Row Constructor algorithm, which virtualizes the table reference. This eliminates contention associated with the use of SYS.DUMMY. Note that DUMMY still appears as the table and/or correlation name in short, long, and graphical plans. See "Row Constructor algorithm" [*SQL Anywhere Server - SQL Usage*].

ISYSARTICLE system table

Each row in the ISYSARTICLE system table describes an article in a publication. See "SYSARTICLE system view" on page 754.

ISYSARTICLECOL system table

Each row in the ISYSARTICLECOL system table identifies a column in an article. See "SYSARTICLECOL system view" on page 755.

ISYSATTRIBUTE system table

This table is for internal use only.

ISYSATTRIBUTENAME system table

This table is for internal use only.

ISYSCAPABILITY system table

Each row in the ISYSCAPABILITY system table identifies a capability of a remote server. See "SYSCAPABILITY system view" on page 755.

ISYSCAPABILITYNAME system table

Each row in the ISYSCAPABILITYNAME system table names a capability that is defined in the ISYSCAPABILITY system table. See "SYSCAPABILITYNAME system view" on page 756.

ISYSCHECK system table

Each row in the ISYSCHECK system table identifies a named check constraint in a table. See "SYSCHECK system view" on page 757.

ISYSCOLPERM system table

Each row in the ISYSCOLPERM system table describes an UPDATE, SELECT, or REFERENCES permission on a column. See "SYSCOLPERM system view" on page 757.

ISYSCOLSTAT system table

The ISYSCOLSTAT system table contains the column statistics used by the optimizer. See "SYSCOLSTAT system view" on page 758.

ISYSCONSTRAINT system table

Each row in the ISYSCONSTRAINT system table describes a named constraint for all tables except the system tables. See "SYSCONSTRAINT system view" on page 759.

ISYSDEPENDENCY system table

Each row in the ISYSDEPENDENCY system table describes a table or view dependency. See "SYSDEPENDENCY system view" on page 760.

ISYSDOMAIN system table

Each of the predefined data types (sometimes called **domains**) is assigned a unique number. The ISYSDOMAIN table is provided for informational purposes, to show the association between these numbers and the appropriate data types. This table is never changed. See "SYSDOMAIN system view" on page 761.

ISYSEVENT system table

Each row in the ISYSEVENT system table describes an event created with CREATE EVENT. See "SYSEVENT system view" on page 761.

ISYSEVENTTYPE system table

The ISYSEVENTTYPE system table defines the system event types that can be referenced by CREATE EVENT. See "SYSEVENTTYPE system view" on page 763

ISYSEXTERNLOGIN system table

Each row in the ISYSEXTERNLOGIN system table describes an external login for remote data access. See "SYSEXTERNLOGIN system view" on page 763.

ISYSFILE system table

Each row in the ISYSFILE system table describes a dbspace for a database. Every database consists of one or more dbspaces; each dbspace corresponds to an operating system file. See "SYSFILE system view" on page 764.

ISYSFKEY system table

Each row in the ISYSFKEY system table describes a foreign key in the database. See "SYSFKEY system view" on page 765.

ISYSGROUP system table

Each row in the ISYSGROUP system table defines a member of a group. This table describes the many-to-many relationship between groups and members. See "SYSGROUP system view" on page 766

ISYSHISTORY system table

Each row in the ISYSHISTORY system table indicates a time in which the database was started with a different version of the software and/or on a different platform. See "SYSHISTORY system view" on page 767.

ISYSIDX system table

Each row in the ISYSIDX system table describes an index in the database. See "SYSIDX system view" on page 768.

ISYSIDXCOL system table

Each row in the ISYSIDXCOL system table describes a column in an index. See "SYSIDXCOL system view" on page 770.

ISYSJAR system table

Each row in the ISYSJAR system table defines a JAR file in the system. See "SYSJAR system view" on page 771.

ISYSJARCOMPONENT system table

Each row in the ISYSJAR system table defines a JAR file component. See "SYSJARCOMPONENT system view" on page 771.

ISYSJAVACLASS system table

Each row in the ISYSJAVACLASS system table describes a Java class. See "SYSJAVACLASS system view" on page 772.

ISYSLOGINMAP system table

The ISYSLOGINMAP system table contains all the User Profile names that can be used to connect to the database using either an integrated login or a Kerberos login. As a security measure, only users with DBA authority can view the contents of this table. See "SYSLOGINMAP system view" on page 773.

ISYSMVOPTION system table

Each row in the ISYSMVOPTION system table describes an option for a materialized view. See "SYSMVOPTION system view" on page 774.

ISYSMVOPTIONNAME system table

Each row in the ISYSMVOPTIONNAME system table provides the name of a materialized view listed in ISYSMVOPTION. See "SYSMVOPTIONNAME system view" on page 774.

ISYSOBJECT system table

Each row in the ISYSOBJECT system view describes an object. Examples of database objects include tables, views, columns, indexes, and procedures. See "SYSOBJECT system view" on page 775.

ISYSOPTION system table

Each row in the ISYSOPTION system table describes the settings for an option for one user ID. Options settings are stored in the ISYSOPTION table by the SET command, and each user can have their own setting for each option. See "SYSOPTION system view" on page 776.

ISYSOPTSTAT system table

The ISYSOPTSTAT system table stores the cost model calibration information as computed by the ALTER DATABASE CALIBRATE statement. See "SYSOPTSTAT system view" on page 777.

ISYSPHYSIDX system table

Each row in the ISYSPHYSIDX system table describes a physical index in the database. See "SYSPHYSIDX system view" on page 777.

ISYSPROCEDURE system table

Each row in the ISYSPROCEDURE system table describes a procedure in the database. See "SYSPROCEDURE system view" on page 779.

ISYSPROCPARM system table

Each row in the ISYSPROCPARM system table describes a parameter to a procedure in the database. See "SYSPROCPARM system view" on page 780.

ISYSPROCPERM system table

Each row in the ISYSPROCPERM system table describes a user granted permission to call one procedure. See "SYSPROCPERM system view" on page 781.

ISYSPROXYTAB system table

Each row in the ISYSPROXYTAB system table describes a proxy table. See "SYSPROXYTAB system view" on page 782.

ISYSPUBLICATION system table

Each row in the ISYSPUBLICATION system table describes a SQL Remote or MobiLink publication. See "SYSPUBLICATION system view" on page 783.

ISYSREMARK system table

Each row in the ISYSREMARK system table describes a remark (or comment) for an object. See "SYSREMARK system view" on page 784.

ISYSREMOTEOPTION system table

Each row in the ISYSREMOTEOPTION system table describes the values of a SQL Remote message link parameter. See "SYSREMOTEOPTION system view" on page 784.

ISYSREMOTEOPTIONTYPE system table

Each row in the ISYSREMOTEOPTIONTYPE system table describes one of the SQL Remote message link parameters. See "SYSREMOTEOPTIONTYPE system view" on page 785.

ISYSREMOTETYPE system table

The ISYSREMOTETYPE system table contains information about SQL Remote. See "SYSREMOTETYPE system view" on page 785.

ISYSREMOTEUSER system table

Each row in the ISYSREMOTEUSER system table describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user. See "SYSREMOTEUSER system view" on page 786.

ISYSSCHEDULE system table

Each row in the ISYSSCHEDULE system table describes a time at which an event is to fire, as specified by the SCHEDULE clause of CREATE EVENT. See "SYSSCHEDULE system view" on page 788.

ISYSSERVER system table

Each row in the ISYSSERVER system table describes a remote server. See "SYSSERVER system view" on page 789.

ISYSSOURCE system table

Each row in the ISYSSOURCE system view contains the source for an object listed in the ISYSOBJECT system table. See "SYSSOURCE system view" on page 790.

ISYSSQLSERVERTYPE system table

The ISYSSQLSERVERTYPE system table contains information relating to compatibility with Adaptive Server Enterprise. See "SYSSQLSERVERTYPE system view" on page 790.

ISYSSUBSCRIPTION system table

Each row in the ISYSSUBSCRIPTION system table describes a subscription from one user ID (which must have REMOTE permissions) to one publication. See "SYSSUBSCRIPTION system view" on page 791.

ISYSSYNC system table

This table contains information relating to MobiLink synchronization. Some columns in this table contain potentially sensitive data. For that reason, access to this table is restricted to users with DBA authority. The

SYSSYNC2 view provides public access to the data in this table except for the potentially sensitive columns. See "SYSSYNC system view" on page 792.

ISYSSYNCSCRIPT system table

This table contains information relating to MobiLink synchronization scripts. See "SYSSYNCSCRIPT system view" on page 793.

ISYSTAB system table

Each row in the ISYSTAB system table describes one table in the database. See "SYSTAB system view" on page 794.

ISYSTABCOL system table

Each row in the ISYSTABCOL system table describes a column of a table in the database. See "SYSTABCOL system view" on page 797.

ISYSTABLEPERM system table

Each row in the ISYSTABLEPERM system table corresponds to one table, one user ID granting the permission (**grantor**) and one user ID granted the permission (**grantee**). See "SYSTABLEPERM system view" on page 799.

ISYSTRIGGER system table

Each row in the ISYSTRIGGER system table describes a trigger in the database. See "SYSTRIGGER system view" on page 800.

ISYSTYPEMAP system table

The ISYSTYPEMAP system table contains the compatibility mapping values for the ISYSSQLSERVERTYPE system table. See "SYSTYPEMAP system view" on page 802.

ISYSUSER system table

Each row in the ISYSUSER system table describes a user in the system. See "SYSUSER system view" on page 803.

ISYSUSERAUTHORITY system table

Each row in the ISYSUSERAUTHORITY system table describes the authority granted to a user. See "SYSUSERAUTHORITY system view" on page 804.

ISYSUSERMESSAGE system table

Each row in the ISYSUSERMESSAGE system table holds a user-defined message for an error condition. See "SYSUSERMESSAGE system view" on page 804.

ISYSUSERTYPE system table

Each row in the ISYSUSERTYPE system table describes a user-defined data type. See "SYSUSERTYPE system view" on page 805.

ISYSVIEW system table

Each row in the ISYSVIEW system table describes a view in the database. See "SYSVIEW system view" on page 806.

ISYSWEBSERVICE system table

Each row in the ISYSWEBSERVICE system table describes a web service. See "SYSWEBSERVICE system view" on page 807.

Diagnostic tracing tables

Following are the main tables that are used for application profiling and diagnostic tracing. These tables are owned by the **dbo** user. For many of these tables, there exists a global shared temporary table with a similar name and schema. For example, the sa_diagnostic_blocking table has a global temporary table counterpart, sa_tmp_diagnostic_blocking table, which has the same schema. During a tracing session, diagnostic data is written to these temporary tables. Because temporary tables are not logged, they provide superior performance during a tracing session, where it is important to minimize the impact on the server.

See also

- "Application profiling" [*SQL Anywhere Server - SQL Usage*]
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_auxiliary_catalog table

The sa_diagnostic_auxiliary_catalog table is owned by the dbo user, and is used to map database objects between the production database and tracing database. Objects include user tables, procedures, and functions. This table is used primarily by the Index Consultant and the TRACED_PLAN function.

Columns

Column name	Column type	Column constraint	Table constraints
original_object_id	UNSIGNED BIGINT	NOT NULL	Primary key.
local_object_id	UNSIGNED BIGINT	NOT NULL	Unique.
pages_if_table	UNSIGNED INT		
rows_if_table	UNSIGNED BIGINT		

original_object_id The object ID of this object in the main tracing database.

local_object_id The object ID of this object in the auxiliary tracing database.

pages_if_table If the object is a table, this is the number of pages in the table. If the object is not a table, this value is NULL.

rows_if_table If the object is a table, this is the number of rows in the table. If the object is not a table, this value is NULL.

See also

- "TRACED_PLAN function [Miscellaneous]" on page 269
- "Index Consultant" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_blocking table

The sa_diagnostic_blocking table is owned by the dbo user, and records blocking events. If logging of blocking events is enabled, a row is inserted in this table each time a connection is blocked while trying to access a resource. Typically, this is caused by either a table or a row lock. A large number of blocks may indicate that you should examine the concurrency in your application to reduce contention for tables and rows.

There are two versions of this table: sa_diagnostic_blocking, and sa_tmp_diagnostic_blocking.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Foreign key references sa_diagnostic_cursor. Foreign key references sa_diagnostic_request.
lock_id	UNSIGNED BIGINT	NOT NULL	
request_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_request.
cursor_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_cursor.
original_table_object_id	UNSIGNED BIGINT		
rowid	UNSIGNED BIGINT		
block_time	TIMESTAMP	NOT NULL	
unblock_time	TIMESTAMP		
blocked_by	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

lock_id The ID of the lock that caused the blocking if a row or table lock caused the block, otherwise NULL.

request_id The ID of the request that was blocked if the block did not occur because of a cursor, otherwise NULL. This value corresponds to the ID assigned to the request in sa_diagnostic_request.

cursor_id The ID of the cursor if the block occurred because of a cursor, otherwise NULL. This value corresponds to the ID assigned to the cursor in sa_diagnostic_cursor.

original_table_object_id If the block occurred because of a table lock, the ID of the table on which the block occurred, otherwise NULL.

rowid If the block occurred because of a row lock, the ID of the row on which the block occurred, otherwise NULL.

block_time The time at which the block occurred.

unblock_time The time at which the block ended.

blocked_by The ID of the connection that held the lock, causing the block.

See also

- "Transaction blocking and deadlock" [*SQL Anywhere Server - SQL Usage*]
- "How locking works" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_cachecontents table

The sa_diagnostic_cachecontents table is owned by the dbo user. When diagnostic tracing is enabled, periodic snapshots of the cache contents are taken. The sa_diagnostic_cachecontents table records the number of table pages for each table in the cache at the time the snapshot was taken, as well as the number of rows in each table. The optimizer can use this information to recreate the conditions under which a query was originally optimized, and then make optimization decisions.

Data in the sa_diagnostic_cachecontents table is updated every 20 seconds, as long as there is query activity.

There are two versions of this table: sa_diagnostic_cachecontents, and sa_tmp_diagnostic_cachecontents.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
"time"	TIMESTAMP	NOT NULL	Primary key.
original_table_object_id	UNSIGNED BIGINT	NOT NULL	Primary key.
pages_in_cache	UNSIGNED INT	NOT NULL	
num_table_pages	UNSIGNED INT	NOT NULL	
num_table_rows	UNSIGNED BIGINT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

"time" The time at which the snapshot of the cache was taken.

original_table_object_id The object ID of each table represented in the snapshot.

pages_in_cache For a specified table in the snapshot, the total number of pages in cache at the moment of the snapshot.

num_table_pages For a specified table in the snapshot, the total number of pages for the table.

num_table_rows For a specified table in the snapshot, the total number of rows in the table.

sa_diagnostic_connection table

The sa_diagnostic_connection table is owned by the dbo user, and has one row for every database connection that is active during the logging session. Connect and disconnect times, if they occur within the logging session, can be derived from the sa_diagnostic_request table.

Most of the values in this table mirror values of connection properties.

There are two versions of this table: sa_diagnostic_connection, and sa_tmp_diagnostic_connection.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key.
connection_number	UNSIGNED INT		Primary key.
connection_name	LONG VARCHAR		
user_name	LONG VARCHAR		
comm_link	CHAR(40)		
node_address	LONG VARCHAR		
appinfo	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

connection_number A number assigned by the database server to identify the user's connection to the database. This value reflects the value of the Number connection property.

connection_name Optional name property for the connection. This value reflects the value of the Name connection property.

user_name The name of the user connected to the database.

comm_link Specifies the client side network protocol options. This value reflects the value of the CommLinks connection property.

node_address The node for the client in a client/server connection. This value reflects the value of the NodeAddress connection property.

appinfo Information about the client process, such as the IP address of the client computer, the operating system it is running on, and so on. This value reflects the value of the AppInfo connection property.

See also

- "Connection-level properties" [*SQL Anywhere Server - Database Administration*]

sa_diagnostic_cursor table

The sa_diagnostic_cursor table is owned by the dbo user. Each row describes either an internal or external cursor opened during the logging session.

There are two versions of this table: sa_diagnostic_cursor, and sa_tmp_diagnostic_cursor.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_query.
cursor_id	UNSIGNED BIGINT	NOT NULL	Primary key.
query_id	UNSIGNED BIGINT	NOT NULL	Foreign key references sa_diagnostic_query.
isolation_level	TINYINT		
flags	UNSIGNED INT		
forward_fetches	UNSIGNED INT		
reverse_fetches	UNSIGNED INT		
absolute_fetches	UNSIGNED INT		
first_fetch_time_ms	UNSIGNED INT		
total_fetch_time_ms	UNSIGNED INT		
plan_xml	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

cursor_id A unique number identifying the cursor.

query_id Identifies the query over which this cursor ranges.

isolation_level Isolation level at which this cursor was opened.

flags Internal use.

forward_fetches Number of forward fetches, including prefetches, done on the cursor.

reverse_fetches Number of reverse fetches, including prefetches, done on the cursor.

absolute_fetches Number of absolute fetches done on the cursor.

first_fetch_time_ms Duration of time spent fetching the first row.

total_fetch_time_ms Duration of time spent fetching. This value does not include application processing time between actual fetches (think time).

plan_xml Detailed plan for cursors that were dumped at the time the cursor was closed. These plans contain detailed statistics where appropriate.

See also

- "Introduction to cursors" [*SQL Anywhere Server - Programming*]

sa_diagnostic_deadlock table

The sa_diagnostic_deadlock table is owned by the dbo user. When diagnostic tracing is enabled and is set to include tracing of deadlock events, a set of rows is inserted into this table every time a deadlock occurs (one row for each connection that was part of the deadlock is inserted). The set of all rows that comprise a single deadlock event is uniquely identified by a snapshot_id.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
snapshot_id	UNSIGNED BIGINT	NOT NULL	
snapshot_at	TIMESTAMP	NOT NULL	
waiter	UNSIGNED INT	NOT NULL	
request_id	UNSIGNED BIGINT		
original_table_object_id	UNSIGNED BIGINT		
rowid	UNSIGNED BIGINT		
owner	UNSIGNED INT	NOT NULL	
rollback_operation_count	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

snapshot_id A number identifying which deadlock event this row is a part of. Note that this column has nothing to do with snapshot isolation.

snapshot_at The time at which the deadlock occurred.

waiter The connection number of the connection that this row represents.

request_id The ID of the request that this connection was processing when the deadlock occurred.

original_table_object_id The object ID of the table on which this connection was blocked.

rowid The record ID of the row on which this connection was blocked.

owner The connection number of the connection that locked the desired row.

rollback_operation_count The number of uncommitted operations.

See also

- "Transaction blocking and deadlock" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_hostvariable table

The sa_diagnostic_hostvariable table is owned by the dbo user, and contains the values of host variables used by the specified cursor.

There are two versions of this table: sa_diagnostic_hostvariable, and sa_tmp_diagnostic_hostvariable.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_request.
request_id	UNSIGNED BIGINT	NOT NULL	Primary key. Foreign key references sa_diagnostic_request.
cursor_id	UNSIGNED BIGINT		
hostvar_num	UNSIGNED SMALL-INT	NOT NULL	Primary key.
hostvar_type	UNSIGNED TINYINT	NOT NULL	
hostvar_value	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

request_id The ID of the request to which the host variables belong.

cursor_id The ID of the cursor to which the host variables pertain.

hostvar_num The ordinal position of the host variable in the SQL statement.

hostvar_type The domain number of the host variable, typically a string, integer, or a float.

hostvar_value A string representing the value of the host variable. Even if the host variable is an integer or a float, the value is still represented here as a string.

See also

- "Using host variables" [*SQL Anywhere Server - Programming*]

sa_diagnostic_internalvariable table

The sa_diagnostic_internalvariable table is owned by the dbo user, and contains the values of internal (local) variables used by a given statement. This table is primarily used by the Index Consultant, and the traced_plan function.

There are two versions of this table: sa_diagnostic_internalvariable, and sa_tmp_diagnostic_internalvariable.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
request_id	UNSIGNED BIGINT		
rowvariable_id	UNSIGNED INT		
variable_domain	UNSIGNED SMALL-INT		
variable_name	CHAR(128)		
variable_value	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

request_id The ID of the request that contains the internal variable.

rowvariable_id The column number in the row variable of this value.

variable_domain The data type of the internal variable.

variable_name The name of the internal variable.

variable_value A string representing the value of the internal variable.

See also

- "Local variables" on page 36

sa_diagnostic_query table

The sa_diagnostic_query table is owned by the dbo user, and stores optimization information for queries, especially the context in which they were optimized. A row in this table represents an invocation of the optimizer for a query. Plans captured at optimization time are stored here.

Some of the values in this table mirror database option values.

There are two versions of this table: sa_diagnostic_query, and sa_tmp_diagnostic_query.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_statement
query_id	UNSIGNED BIGINT	NOT NULL	Primary key. Foreign key references sa_diagnostic_statement.
statement_id	UNSIGNED BIGINT	NOT NULL	
user_object_id	UNSIGNED BIGINT	NOT NULL	
start_time	TIMESTAMP	NOT NULL	
cache_size_bytes	UNSIGNED BIGINT		
optimization_goal	TINYINT		
optimization_level	TINYINT		
user_estimates	TINYINT		
optimization_workload	TINYINT		
available_requests	TINYINT		
active_requests	TINYINT		
max_tasks	TINYINT		
used_bypass	TINYINT		
estimated_cost_ms	TINYINT		
plan_explain	LONG VARCHAR		
plan_xml	LONG VARCHAR		
sql_rewritten	LONG VARCHAR		

logging_session_id The ID of the logging session during which the query or request occurred.

query_id A number uniquely identifying the query.

statement_id A number uniquely identifying a statement in a query.

user_object_id The object ID of the user under which this query was executed. If the query was run from a procedure, this would be the user ID of the procedure owner.

start_time The time at which this query was optimized.

cache_size_bytes The size, in bytes, of the cache at the time this query was optimized.

optimization_goal Determines whether query processing is optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set. This value reflects the value of the optimization_goal database option.

To see possible values for this column, see "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].

optimization_level Controls the amount of effort made by the SQL Anywhere query optimizer to find an access plan for a SQL statement. This value reflects the value of the optimization_level database option.

To see possible values for this column, see "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*].

user_estimates Controls whether or not user selectivity estimates in query predicates are respected or ignored by the query optimizer. This value reflects the value of the user_estimates database option.

To see possible values for this column, see "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*].

optimization_workload Determines whether query processing is optimized towards a workload that is a mix of updates and reads or a workload that is predominantly read-based. This value reflects the value of the optimization_workload database option.

To see possible values for this column, see "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

available_requests Used internally to compute the level of intra-query parallelism.

active_requests Used internally to compute the level of intra-query parallelism.

max_tasks Used internally to compute the level of intra-query parallelism.

used_bypass Whether a simple query bypass was used. A value of 1 indicates a bypass was used; a value of 0 indicates that the query was fully optimized.

estimated_cost_ms The estimated cost, in milliseconds.

plan_explain A text plan representation of this query.

plan_xml A graphical plan representation of the query (if one was recorded).

sql_rewritten Text of a query after applying optimizations. A value will only be present in this column if optimization logging is enabled.

See also

- "Database Options" [*SQL Anywhere Server - Database Administration*]
- "How the optimizer works" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_request table

The sa_diagnostic_request table is owned by the dbo user, and is the master table for all requests. A request is an event related to query processing and generally includes:

- connect or disconnect events
- statement executions
- statement preparations
- open or drop cursor events

There are two versions of this table: sa_diagnostic_request and sa_tmp_diagnostic_request.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_connection. Foreign key references sa_diagnostic_cursor. Foreign key references sa_diagnostic_query. Foreign key references sa_diagnostic_statement.
request_id	UNSIGNED BIGINT	NOT NULL	Primary key.
start_time	TIMESTAMP	NOT NULL	
finish_time	TIMESTAMP	NOT NULL	
duration_ms	UNSIGNED INT	NOT NULL	
connection_number	UNSIGNED INT		Foreign key references sa_diagnostic_connection.
request_type	UNSIGNED SMALLINT		

Column name	Column type	Column constraint	Table constraints
statement_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_statement.
query_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_query.
cursor_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_cursor.
sql_code	SMALLINT		

logging_session_id The logging session during which the request occurred.

request_id A number uniquely identifying the request.

start_time The time at which the event started.

finish_time For statement execution, the time when the statement completed; otherwise, NULL.

duration_ms The duration of the event in milliseconds.

connection_number The ID of the connection that caused the event to happen.

request_type The type of request. Values include:

Value	Description
1	Start of new tracing session
2	Statement execution
3	Cursor open
4	Cursor close
5	Connect
6	Disconnect

statement_id If the event was statement-related, the ID assigned to the statement for tracing purposes.

query_id If the event was query-related, the ID assigned to the query for tracing purposes.

cursor_id If the event was cursor-related, the ID assigned to the cursor for tracing purposes.

sql_code Since rows in this table represent operations on statements, cursors, or queries, most return a SQL code. This column contains the SQL code returned. If a SQL code of 0 is returned, the column contains NULL.

sa_diagnostic_statement table

The sa_diagnostic_statement table is owned by the dbo user, and stores the text of statements. A row in this table represents a SQL statement that was executed by the server. Such statements may have been issued by an external source, such as a client request, or by an internal source such as a procedure, trigger, or user-defined function. Internal statements only appear here once per session.

There are two versions of this table: sa_diagnostic_statement, and sa_tmp_diagnostic_statement.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key.
statement_id	UNSIGNED BIGINT	NOT NULL	Primary key.
database_object	UNSIGNED BIGINT		
line_number	UNSIGNED SMALL-INT		
signature	UNSIGNED INT		
statement_text	LONG VARCHAR	NOT NULL	

logging_session_id The logging session during which the statement was submitted.

statement_id A unique number assigned to the statement for tracing purposes.

database_object If the statement came from a procedure, trigger, or function, this is the ID as specified in the ISYSOBJECT system table.

line_number If the statement formed part of a compound statement, this reflects the ordinal position of the statement within the compound statement.

signature Used internally to group similar queries.

statement_text The statement text.

sa_diagnostic_statistics table

The sa_diagnostic_statistics table is owned by the dbo user, and contains a history of performance counters maintained in the server. Each row represents the value of a given performance counter at a given moment in time.

There are two versions of this table: sa_diagnostic_statistics, and sa_tmp_diagnostic_statistics.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
"time"	TIMESTAMP	NOT NULL	
counter_id	UNSIGNED SMALL-INT	NOT NULL	
type	TINYINT	NOT NULL	
connection_number	UNSIGNED INT	NOT NULL	
counter_value	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

"time" The time at which the performance counter value was captured.

counter_id A number uniquely identifying the performance counter. You can get the name of the property that this counter_id represents using the PROPERTY_NAME function.

type Indicates whether this is a database, server, or connection statistic. Possible values are 0 for server, 1 for database, 2 for connection, and 4 for external database.

connection_number In the case of a connection statistic, the connection number from which this property was captured. In the case of an extended database statistic, the file number for the file from which this property was captured. Otherwise, the value is 0.

counter_value The value of the performance counter.

See also

- "PROPERTY_NAME function [System]" on page 217

sa_diagnostic_tracing_level table

The sa_diagnostic_tracing_level table is owned by the dbo user, and each row in this table is a condition that determines what kind of diagnostic information to send to the tracing database. If a piece of logging data meets the conditions of one or more rows in this table, then the corresponding data is logged.

Data in this table is populated using the CONNECT TRACING or REFRESH TRACING LEVELS statements.

Columns

Column name	Column type	Column constraint	Table constraints
id	UNSIGNED INT	NOT NULL	Primary key.

Column name	Column type	Column constraint	Table constraints
scope	CHAR(32)	NOT NULL	
identifier	CHAR(128)		
trace_type	CHAR(32)	NOT NULL	
trace_condition	CHAR(32)		
value	UNSIGNED INT		
enabled	BIT	NOT NULL	

scope The scope of the diagnostic tracing, as listed below. To see the description for each scope, see "Diagnostic tracing scopes" [*SQL Anywhere Server - SQL Usage*].

- DATABASE
- ORIGIN
- USER
- CONNECTION_NAME
- CONNECTION_NUMBER
- FUNCTION
- PROCEDURE
- EVENT
- TRIGGER
- TABLE

id For internal use only.

identifier The identifier for the scope. This value changes, depending on the specified scope. For example:

- if *scope* is DATABASE, *identifier* may not be present.
- if *scope* is ORIGIN, *identifier* must be either Internal or External.
- if *scope* is USER, *identifier* is the ID of the user.
- if *scope* is CONNECTION_NAME, or CONNECTION_NUMBER, *identifier* is the name or number, respectively, for the connection.
- if *scope* is FUNCTION, PROCEDURE, EVENT, TRIGGER, or TABLE, *identifier* is the fully qualified identifier for the object.

trace_type The type of data to trace for the specified scope, as listed below. To see the description for each trace type, see "Diagnostic tracing types" [*SQL Anywhere Server - SQL Usage*].

- VOLATILE_STATISTICS
- NONVOLATILE_STATISTICS
- CONNECTION_STATISTICS
- BLOCKING

- PLANS
- PLANS_WITH_STATISTICS
- STATEMENTS
- STATEMENTS_WITH_VARIABLES
- OPTIMIZATION_LOGGING
- OPTIMIZATION_LOGGING_WITH_PLANS

condition Applies only to plans, and controls whether to trace large, expensive queries, or queries for which the optimizer did not make optimal choices. Possible values are listed below. To see a description of each condition, see "Diagnostic tracing conditions" [*SQL Anywhere Server - SQL Usage*].

- NONE, or NULL
- SAMPLE_EVERY
- ABSOLUTE_COST
- RELATIVE_COST_DIFFERENCE

condition_value The value associated with the *condition*. For example, if *condition* is SAMPLE_EVERY, the *condition_value* would be a positive integer reflecting time in milliseconds. Additional rules are as follows:

- If *condition* is NULL or NONE, there is no *condition_value*.
- If *condition* is ABSOLUTE_COST, *condition_value* reflects the difference between expected and real cost of executing, in milliseconds.
- If *condition* is RELATIVE_COST_DIFFERENCE, *condition_value* reflects the cost of executing, as a percentage of the estimated cost.

enabled Whether the row is enabled. That is, whether the tracing settings in the row are active. 1 is enabled; 0 is disabled.

See also

- "ATTACH TRACING statement" on page 344
- "REFRESH TRACING LEVEL statement" on page 623

Other tables

Following is information about other tables such as system tables used by Java in the database and SQL Remote.

RowGenerator table (dbo)

The dbo.RowGenerator table is provided as a read-only table that has 255 rows. This table can be useful for queries which produce small result sets and which need a range of numeric values.

The RowGenerator table is used by system procedures and views, and should not be modified in any way.

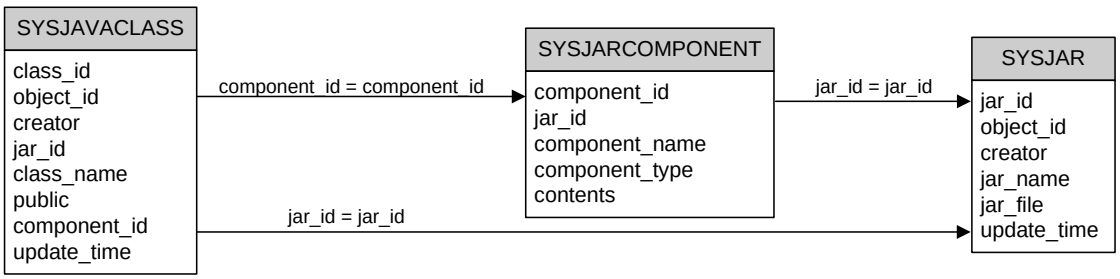
You can also use the sa_rowgenerator system procedure to generate a range of numeric values. For more information on using the sa_rowgenerator system procedure, including examples, see "sa_rowgenerator system procedure" on page 910.

Column name	Column type	Column constraint	Underlying table constraints
row_num	SMALLINT	NOT NULL	

row_num A value between 1 and 255.

Java system tables

The system tables that are used for Java are listed below. Foreign key relations between tables are indicated by arrows: the arrow leads from the foreign table to the primary table.



MobiLink system tables

For information about the MobiLink system tables, see "MobiLink Server System Tables" [*MobiLink - Server Administration*].

SQL Remote system tables

For information about the SQL Remote system tables, see "SQL Remote system tables" [*SQL Remote*].

UltraLite system tables

For information about the UltraLite system tables, see "UltraLite system tables" [*UltraLite - Database Management and Reference*].

CHAPTER 6

Views

Contents

System views in Sybase Central 754
Consolidated views 809
Compatibility views 824

System views in Sybase Central

The catalog contains system tables that link together by keys and indexes. In SQL Anywhere, the system tables are hidden. However, there is a system view for each table. In some cases, a system view may also include columns from more than one system tables, in order to satisfy a commonly needed join.

To ensure compatibility with future versions of the SQL Anywhere catalogue, make sure your applications make use of system views and not the underlying system tables, which may change.

Detailed information about system views, including the view definition, is available in Sybase Central:

- To view system views, right-click a connected database, choose Filter Objects by Owner, and select SYS. Open the Views folder for the database.
- You can see the view definition by selecting the view in the left pane and then clicking the SQL tab in the right pane.
- To display the data, open the View folder in the left pane and select a view. In the right pane, click the Data tab.

SYSARTICLE system view

Each row of the SYSARTICLE system view describes an article in a publication. The underlying system table for this view is ISYSARTICLE.

Columns

Column name	Column type	Column constraint
publication_id	UNSIGNED INT	NOT NULL
table_id	UNSIGNED INT	NOT NULL
where_expr	LONG VARCHAR	
subscribe_by_expr	LONG VARCHAR	
query	CHAR(1)	NOT NULL
alias	VARCHAR(256)	

publication_id The publication of which the article is a part.

table_id Each article consists of columns and rows from a single table. This column contains the table ID for this table.

where_expr For articles that contain a subset of rows defined by a WHERE clause, this column contains the search condition.

subscribe_by_expr For articles that contain a subset of rows defined by a SUBSCRIBE BY expression, this column contains the expression.

query Indicates information about the article type to the database server.

alias The alias for the article.

Constraints on underlying system table

PRIMARY KEY (publication_id, table_id)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

SYSARTICLECOL system view

Each row of the SYSARTICLECOL system view identifies a column in an article. The underlying system table for this view is ISYSARTICLECOL.

Columns

Column name	Column type	Column constraint
publication_id	UNSIGNED INT	NOT NULL
table_id	UNSIGNED INT	NOT NULL
column_id	UNSIGNED INT	NOT NULL

publication_id A unique identifier for the publication of which the column is a part.

table_id The table to which the column belongs.

column_id The column identifier, from the SYSTABCOL system view.

Constraints on underlying system table

PRIMARY KEY (publication_id, table_id, column_id)

FOREIGN KEY (publication_id, table_id) references SYS.ISYSARTICLE (publication_id, table_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

SYSCAPABILITY system view

Each row of the SYSCAPABILITY system view identifies a capability of a remote server. The underlying system table for this view is ISYSCAPABILITY.

Columns

Column name	Column type	Column constraint
capid	INTEGER	NOT NULL

Column name	Column type	Column constraint
srvid	UNSIGNED INT	NOT NULL
capvalue	CHAR(128)	NOT NULL

capid The ID of the capability, as listed in the SYSCAPABILITYNAME system view.

srvid The server to which the capability applies, as listed in the SYSSERVER system view.

capvalue The value of the capability.

Constraints on underlying system table

PRIMARY KEY (capid, srvid)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

FOREIGN KEY (capid) references SYS.ISYSCAPABILITYNAME (capid)

See also

- "SYSCAPABILITYNAME system view" on page 756

SYSCAPABILITYNAME system view

Each row in the SYSCAPABILITYNAME system view names a capability that is defined in the SYSCAPABILITY system view. The underlying system table for this view is ISYSCAPABILITYNAME.

Columns

Column name	Column type	Column constraint
capid	INTEGER	NOT NULL
capname	CHAR(128)	NOT NULL

capid A number uniquely identifying the capability.

capname The name of the capability.

Constraints on underlying system table

PRIMARY KEY (capid)

See also

- "SYSCAPABILITY system view" on page 755

SYSCHECK system view

Each row in the SYSCHECK system view provides the definition for a named check constraint in a table. The underlying system table for this view is ISYSCHECK.

Columns

Column name	Column type	Column constraint
check_id	UNSIGNED INT	NOT NULL
check_defn	LONG VARCHAR	NOT NULL

check_id A number that uniquely identifies the constraint in the database.

check_defn The CHECK expression.

Constraints on underlying system table

PRIMARY KEY (check_id)

FOREIGN KEY (check_id) references SYS.ISYSCONSTRAINT (constraint_id)

SYSCOLPERM system view

The GRANT statement can give UPDATE, SELECT, or REFERENCES permission to individual columns in a table. Each column with UPDATE, SELECT, or REFERENCES permission is recorded in one row of the SYSCOLPERM system view. The underlying system table for this view is ISYSCOLPERM.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
grantee	UNSIGNED INT	NOT NULL
grantor	UNSIGNED INT	NOT NULL
column_id	UNSIGNED INT	NOT NULL
privilege_type	SMALLINT	NOT NULL
is_grantable	CHAR(1)	NOT NULL

table_id The table number for the table containing the column.

grantee The user number of the user ID that is given permission on the column. If the grantee is the user number for the special PUBLIC user ID, the permission is given to all user IDs.

grantor The user number of the user ID that grants the permission.

column_id This column number, together with the table_id, identifies the column for which permission has been granted.

privilege_type The number in this column indicates the kind of column permission (16=REFERENCES, 1=SELECT, or 8=UPDATE).

is_grantable (Y/N) Indicates if the permission on the column was granted WITH GRANT OPTION.

Constraints on underlying system table

PRIMARY KEY (table_id, grantee, grantor, column_id, privilege_type)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

FOREIGN KEY (grantor) references SYS.ISYSUSER (user_id)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

SYSCOLSTAT system view

The SYSCOLSTAT system view contains the column statistics, including histograms, that are used by the optimizer. The contents of this view are best retrieved using the sa_get_histogram stored procedure or the Histogram utility. The underlying system table for this view is ISYSCOLSTAT.

NOTE
If the database is encrypted, or if table encryption is enabled, the underlying system table, ISYSCOLSTAT, is also encrypted since it contains histogram information that could reveal underlying data.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
column_id	UNSIGNED INT	NOT NULL
format_id	SMALLINT	NOT NULL
update_time	TIMESTAMP	NOT NULL
density	FLOAT	NOT NULL
max_steps	SMALLINT	NOT NULL
actual_steps	SMALLINT	NOT NULL
step_values	LONG BINARY	
frequencies	LONG BINARY	

table_id A number that uniquely identifies the table or materialized view to which the column belongs.

column_id A number that, together with table_id, uniquely identifies the column.

format_id For system use only.

update_time The time of the last update of the column statistics.

density An estimate of the average selectivity of a single value for the column, not counting the large single value selectivities stored in the row.

max_steps For system use only.

actual_steps For system use only.

step_values For system use only.

frequencies For system use only.

Constraints on underlying system table

PRIMARY KEY (table_id, column_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

SYSCONSTRAINT system view

Each row in the SYSCONSTRAINT system view describes a named constraint in the database. The underlying system table for this view is ISYSCONSTRAINT.

Columns

Column name	Column type	Column constraint
constraint_id	UNSIGNED INT	NOT NULL
constraint_type	CHAR(1)	NOT NULL
ref_object_id	UNSIGNED BIGINT	NOT NULL
table_object_id	UNSIGNED BIGINT	NOT NULL
constraint_name	CHAR(128)	NOT NULL

constraint_id The unique ID for the constraint.

constraint_type The type of constraint:

- C - column check constraint.
- T - table constraint.
- P - primary key.
- F - foreign key.
- U - unique constraint.

ref_object_id The object ID of the column, table, or index to which the constraint applies.

table_object_id The table ID of the table to which the constraint applies.

constraint_name The name of the constraint.

Constraints on underlying system table

PRIMARY KEY (constraint_id)

FOREIGN KEY (ref_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (table_object_id) references SYS.ISYSOBJECT (object_id)

UNIQUE (table_object_id, constraint_name)

SYSDEPENDENCY system view

Each row in the SYSDEPENDENCY system view describes a dependency between two database objects. The underlying system table for this view is ISYSDEPENDENCY.

A dependency exists between two database objects when one object references another object in its definition. For example, if the query specification for a view references a table, the view is said to be dependent on the table. The database server tracks dependencies of views on tables, views, materialized views, and columns.

Columns

Column name	Column type	Column constraint
ref_object_id	UNSIGNED BIGINT	NOT NULL
dep_object_id	UNSIGNED BIGINT	NOT NULL

ref_object_id The object ID of the referenced object.

dep_object_id The ID of the referencing object.

Constraints on underlying system table

PRIMARY KEY (ref_object_id, dep_object_id)

FOREIGN KEY (ref_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (dep_object_id) references SYS.ISYSOBJECT (object_id)

See also

- "sa_dependent_views system procedure" on page 859
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]

SYSDOMAIN system view

The SYSDOMAIN system view records information about built-in data types (also called domains). The contents of this view does not change during normal operation. The underlying system table for this view is ISYSDOMAIN.

Columns

Column name	Column type	Column constraint
domain_id	SMALLINT	NOT NULL
domain_name	CHAR(128)	NOT NULL
type_id	SMALLINT	NOT NULL
"precision"	SMALLINT	

domain_id The unique number assigned to each data type. These numbers cannot be changed.

domain_name The name of the data type normally found in the CREATE TABLE command, such as CHAR or INTEGER.

type_id The ODBC data type. This value corresponds to the value for data_type in the Transact-SQL-compatibility dbo.SYSTYPES table.

"precision" The number of significant digits that can be stored using this data type. The column value is NULL for non-numeric data types.

Constraints on underlying system table

PRIMARY KEY (domain_id)

SYSEVENT system view

Each row in the SYSEVENT system view describes an event created with CREATE EVENT. The underlying system table for this view is ISYSEVENT.

Columns

Column name	Column type	Column constraint
event_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
event_name	VARCHAR(128)	NOT NULL
enabled	CHAR(1)	NOT NULL

Column name	Column type	Column constraint
location	CHAR(1)	NOT NULL
event_type_id	UNSIGNED INT	
action	LONG VARCHAR	
external_action	LONG VARCHAR	
condition	LONG VARCHAR	
remarks	LONG VARCHAR	
source	LONG VARCHAR	

event_id The unique number assigned to each event.

object_id The internal ID for the event, uniquely identifying it in the database.

creator The user number of the owner of the event. The name of the user can be found by looking in the SYSUSER system view.

event_name The name of the event.

enabled (Y/N) Indicates whether or not the event is allowed to fire.

location The location where the event is to fire:

- C = consolidated
- R = remote
- A = all

event_type_id For system events, the event type as listed in the SYSEVENTTYPE system view.

action The event handler definition.

external_action For system use only.

condition The condition used to control firing of the event handler.

remarks Remarks for the event; this column comes from ISYSREMARK.

source The original source for the event; this column comes from ISYSSOURCE.

Constraints on underlying system table

PRIMARY KEY (event_id)

FOREIGN KEY (event_type_id) references SYS.ISYSEVENTTYPE (event_type_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

UNIQUE (index_name, table_id, index_category)

See also

- "SYSEVENTTYPE system view" on page 763

SYSEVENTTYPE system view

The SYSEVENTTYPE system view defines the system event types that can be referenced by CREATE EVENT. The underlying system table for this view is ISYSEVENTTYPE.

Columns

Column name	Column type	Column constraint
event_type_id	UNSIGNED INT	NOT NULL
name	VARCHAR(128)	NOT NULL
description	LONG VARCHAR	

event_type_id The unique number assigned to each event type.

name The name of the system event type.

description A description of the system event type.

Constraints on underlying system table

PRIMARY KEY (event_type_id)

UNIQUE (name)

See also

- "SYSEVENT system view" on page 761

SYSEXTERNLOGIN system view

Each row in the SYSEXTERNLOGIN system view describes an external login for remote data access. The underlying system table for this view is ISYSEXTERNLOGIN.

Note
Previous versions of the catalog contained a SYSEXTERNLOGINS system table. That table has been renamed to be ISYSEXTERNLOGIN (without an 'S'), and is the underlying table for this view.

Columns

Column name	Column type	Column constraint
user_id	UNSIGNED INT	NOT NULL

Column name	Column type	Column constraint
srvid	UNSIGNED INT	NOT NULL
remote_login	VARCHAR(128)	
remote_password	VARBINARY(128)	

user_id The user ID on the local database.

srvid The remote server, as listed in the SYSSERVER system view.

remote_login The login name for the user, for the remote server.

remote_password The password for the user, for the remote server.

Constraints on underlying system table

PRIMARY KEY (user_id, srvid)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

SYSFILE system view

Each row in the SYSFILE system view describes a dbspace for a database. Every database consists of one or more dbspaces; each dbspace corresponds to an operating system file. The underlying system table for this view is ISYSFILE.

SQL Anywhere automatically creates dbspaces for the main database file, temporary file, transaction log file, and transaction log mirror file, but Information about the temporary file, transaction log, and transaction log mirror dbspaces does not appear in the SYSFILE system view. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Columns

Column name	Column type	Column constraint
file_id	SMALLINT	NOT NULL
file_name	LONG VARCHAR	NOT NULL
dbspace_name	CHAR(128)	NOT NULL
store_type	INTEGER	

file_id Each file in a database is assigned a unique number. The SYSTEM dbspace contains all system objects and has a file_id of 0.

file_name The file name for the dbspace. For the SYSTEM dbspace, the value is the name of the database file when the database was created and is for informational purposes only; it cannot be changed. For other dbspaces, the file name can be changed using the following statement:

```
ALTER DBSPACE dbspace RENAME 'new-file-name'
```

dbspace_name A unique name for the dbspace. It is used in the CREATE TABLE command.

store_type This field is for internal use.

Constraints on underlying system table

PRIMARY KEY (file_id)

SYSFKEY system view

Each row in the SYSFKEY system view describes a foreign key constraint in the system. The underlying system table for this view is ISYSFKEY.

Columns

Column name	Column type	Column constraint
foreign_table_id	UNSIGNED INT	NOT NULL
foreign_index_id	UNSIGNED INT	NOT NULL
primary_table_id	UNSIGNED INT	NOT NULL
primary_index_id	UNSIGNED INT	NOT NULL
match_type	TINYINT	NOT NULL
check_on_commit	CHAR(1)	NOT NULL
nulls	CHAR(1)	NOT NULL

foreign_table_id The table number of the foreign table.

foreign_index_id The index number for the foreign key.

primary_table_id The table number of the primary table.

primary_index_id The index number of the primary key.

match_type The matching type for the constraint. Matching types include:

Value	Match type
0	Use the default matching
1	SIMPLE

Value	Match type
2	FULL
129	SIMPLE UNIQUE
130	FULL UNIQUE

For more information on match types, see the MATCH clause of the "CREATE TABLE statement" on page 450.

check_on_commit (Y/N) Indicates whether INSERT and UPDATE statements should wait until the COMMIT to check if foreign keys are still valid.

nulls (Y/N) Indicates whether the columns in the foreign key are allowed to contain the NULL value. Note that this setting is independent of the nulls setting in the columns contained in the foreign key.

Constraints on underlying system table

PRIMARY KEY (foreign_table_id, foreign_index_id)

FOREIGN KEY (foreign_table_id, foreign_index_id) references SYS.ISYSIDX (table_id, index_id)

FOREIGN KEY (primary_table_id, primary_index_id) references SYS.ISYSIDX (table_id, index_id)

SYSGROUP system view

There is one row in the SYSGROUP system view for each member of each group. This view describes the many-to-many relationship between groups and members. A group may have many members, and a user may be a member of many groups. The underlying system table for this view is ISYSGROUP.

Columns

Column name	Column type	Column constraint
group_id	UNSIGNED INT	NOT NULL
group_member	UNSIGNED INT	NOT NULL

group_id The user number of the group.

group_member The user number of a member.

Constraints on underlying system table

PRIMARY KEY (group_id, group_member)

FOREIGN KEY (group_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY (group_member) references SYS.ISYSUSER (user_id)

SYSHISTORY system view

Each row in the SYSHISTORY system view records a system operation on the database, such as a database start, a database calibration, and so on. The underlying system table for this view is ISYSHISTORY.

Columns

Column name	Column type	Column constraint
operation	CHAR(128)	NOT NULL
object_id	UNSIGNED INT	NOT NULL
sub_operation	CHAR(128)	NOT NULL
version	CHAR(128)	NOT NULL
platform	CHAR(128)	NOT NULL
first_time	TIMESTAMP	NOT NULL
last_time	TIMESTAMP	NOT NULL
details	LONG VARCHAR	

operation The type of operation performed on the database file. The operation must be one of the following values:

- **INIT** Information about when the database was created.
- **UPGRADE** Information about when the database was upgraded.
- **START** Information about when the database was started using a specific version of the database server on a particular operating system.
- **LAST_START** Information about the most recent time the database server was started.

  A LAST_START operation is converted to a START operation when the database is started with a different version of the database server and/or on a different operating system than those values currently stored in the LAST_START row.
- **DTT** Information about the *second to last* Disk Transfer Time (DTT) calibration operation performed on the dbspace. That is, information on the second to last execution of either an ALTER DATABASE CALIBRATE or ALTER DATABASE RESTORE DEFAULT CALIBRATION statement.
- **LAST_DTT** Information about the *most recent* DTT calibration operation performed on the dbspace. That is, information on the most recent execution of either an ALTER DATABASE CALIBRATE or ALTER DATABASE RESTORE DEFAULT CALIBRATION statement.
- **LAST_BACKUP** Information about the last backup, including date and time of the backup, the backup type, the files that were backed up, and the version of database server that performed the backup.

object_id For any operation other than DTT and LAST_DTT, the value in this column will be 0. For DTT and LAST_DTT operations, this is the file_id of the dbspace as defined in the SYSFILE system view.

sub_operation For any operation other than DTT and LAST_DTT, the value in this column will be a set of empty single quotes (''). For DTT and LAST_DTT operations, this column contains the type of sub-operation performed on the dbspace. Values include:

- **DTT_SET** The dbspace calibration has been set.
- **DTT_UNSET** The dbspace calibration has been restored to the default setting.

version The version and build number of the database server used to carry out the operation.

platform The operating system on which the operation was carried out.

first_time The date and time the database was first started on a particular operating system with a particular version of the software.

last_time The most recent date and time the database was started on a particular operating system with a particular version of the software.

details This column stores information such as command line options used to start the database server or the capability bits enabled for the database. This information is for use by technical support.

Constraints on underlying system table

PRIMARY KEY (operation, object_id, version, platform)

SYSIDX system view

Each row in the SYSIDX system view defines a logical index in the database. The underlying system table for this view is ISYSIDX.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
index_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
phys_index_id	UNSIGNED INT	
file_id	SMALLINT	NOT NULL
index_category	TINYINT	NOT NULL
"unique"	TINYINT	NOT NULL
index_name	CHAR(128)	NOT NULL

Column name	Column type	Column constraint
not_enforced	CHAR(1)	NOT NULL

table_id Uniquely identifies the table to which this index applies.

index_id A unique number identifying the index within its table.

object_id The internal ID for the index, uniquely identifying it in the database.

phys_index_id Identifies the underlying physical index used to implement the logical index. This value is NULL for indexes on temporary tables or remote tables. Otherwise, the value corresponds to the object_id of a physical index in the SYSPHYSIDX system view. See "SYSPHYSIDX system view" on page 777.

file_id The ID of the file in which the index is contained. This value corresponds to an entry in the SYSFILE system view. See "SYSFILE system view" on page 764.

index_category The type of index. Values include:

Value	Index type
1	Primary key
2	Foreign key
3	Secondary index (includes unique constraints)

"unique" Indicates whether the index is a unique index (1), a non-unique index (4), or a unique constraint (2). A unique index prevents two rows in the indexed table from having the same values in the index columns.

index_name The name of the index.

not_enforced For system use only.

Constraints on underlying system table

PRIMARY KEY (table_id, index_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (table_id, phys_index_id) references SYS.ISYSPHYSIDX (table_id, phys_index_id)

See also

- "SYSIDXCOL system view" on page 770
- "SYSPHYSIDX system view" on page 777
- "SYSFILE system view" on page 764

SYSIDXCOL system view

Each row in the SYSIDXCOL system view describes one column of an index described in the SYSIDX system view. The underlying system table for this view is ISYSIDXCOL.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
index_id	UNSIGNED INT	NOT NULL
sequence	SMALLINT	NOT NULL
column_id	UNSIGNED INT	NOT NULL
"order"	CHAR(1)	NOT NULL
primary_column_id	UNSIGNED INT	

table_id Identifies the table to which the index applies.

index_id Identifies the index to which the column applies. Together, table_id and index_id identify one index described in the SYSIDX system view.

sequence Each column in an index is assigned a unique number starting at 0. The order of these numbers determines the relative significance of the columns in the index. The most important column has sequence number 0.

column_id Identifies which column of the table is indexed. Together, table_id and column_id identify one column described in the SYSCOLUMN system view.

order (A/D) Indicates whether the column in the index is kept in ascending(A) or descending(D) order.

primary_column_id The ID of the primary key column that corresponds to this foreign key column. The value is NULL for non foreign key columns.

Constraints on underlying system table

PRIMARY KEY (table_id, index_id, column_id)

FOREIGN KEY (table_id, index_id) references SYS.ISYSIDX (table_id, index_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

See also

- "SYSIDX system view" on page 768

SYSJAR system view

Each row in the SYSJAR system view defines a JAR file stored in the database. The underlying system table for this view is ISYSJAR.

Columns

Column name	Column type	Column constraint
jar_id	INTEGER	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
jar_name	LONG VARCHAR	NOT NULL
jar_file	LONG VARCHAR	
update_time	TIMESTAMP	NOT NULL

jar_id A unique number identifying the JAR file.

object_id The internal ID for the JAR file, uniquely identifying it in the database.

creator The user number of the creator of the JAR file.

jar_name The name of the JAR file.

jar_file The external file name of the JAR file within the database.

update_time The time the JAR file was last updated.

Constraints on underlying system table

PRIMARY KEY (jar_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

See also

- "SYSJARCOMPONENT system view" on page 771

SYSJARCOMPONENT system view

Each row in the SYSJAR system view defines a JAR file component. The underlying system table for this view is ISYSJARCOMPONENT.

Columns

Column name	Column type	Column constraint
component_id	INTEGER	NOT NULL

Column name	Column type	Column constraint
jar_id	INTEGER	
component_name	LONG VARCHAR	
component_type	CHAR(1)	
contents	LONG BINARY	

component_id The primary key containing the id of the component.

jar_id A field containing the ID number of the JAR.

component_name The name of the component.

component_type The type of the component.

contents The byte code of the JAR file.

Constraints on underlying system table

PRIMARY KEY (component_id)

FOREIGN KEY (jar_id) references SYS.ISYSJAR (jar_id)

See also

- "SYSJAR system view" on page 771

SYSJAVACLASS system view

Each row in the SYSJAVACLASS system view describes one Java class stored in the database. The underlying system table for this view is ISYSJAVACLASS.

Columns

Column name	Column type	Column constraint
class_id	INTEGER	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
jar_id	INTEGER	
class_name	LONG VARCHAR	NOT NULL
public	CHAR(1)	NOT NULL
component_id	INTEGER	
update_time	TIMESTAMP	NOT NULL

Column name	Column type	Column constraint
class_descriptor	LONG BINARY	

class_id The unique number for the Java class. Also the primary key for the table.

object_id The internal ID for the Java class, uniquely identifying it in the database.

creator The user number of the creator of the class.

jar_id The id of the JAR file from which the class came.

class_name The name of the Java class.

public Indicates whether the class is public (Y) or private (N).

component_id The id of the component in the SYSJARCOMPONENT system view.

update_time The last update time of the class.

class_descriptor Not used.

Constraints on underlying system table

PRIMARY KEY (class_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (component_id) references SYS.ISYSJARCOMPONENT (component_id)

SYSLOGINMAP system view

The SYSLOGINMAP system view contains one row for each user that can connect to the database using either an integrated login, or Kerberos login. As a security measure, only users with DBA authority can view the contents of this view. The underlying system table for this view is ISYSLOGINMAP.

Columns

Column name	Column type	Column constraint
login_mode	TINYINT	NOT NULL
login_id	VARCHAR(1024)	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
database_uid	UNSIGNED INT	NOT NULL

login_mode The type of login: 1 for integrated logins, 2 for Kerberos logins.

login_id Either the integrated login user profile name, or the Kerberos principal that maps to database_uid.

object_id A unique identifier, one for each mapping between user ID and database user ID.

database_uid The database user ID to which the login ID is mapped.

Constraints on underlying system table

PRIMARY KEY (login_mode, login_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (database_uid) references SYS.ISYSUSER (user_id)

SYSMVOPTION system view

Each row in the SYSMVOPTION system view describes the setting of one option value for a materialized view at the time of its creation. However, the description does not contain the option name for the option. The underlying system table for this view is ISYSMVOPTION.

Columns

Column name	Column type	Column constraint
view_object_id	UNSIGNED BIGINT	NOT NULL
option_id	UNSIGNED INT	NOT NULL
option_value	LONG VARCHAR	NOT NULL

view_object_id The object ID of the materialized view.

option_id A unique number identifying the option in the database. To see the option name, see the SYSMVOPTIONNAME system view.

option_value The value of the option at the time that the materialized view was created.

Constraints on underlying system table

PRIMARY KEY (view_object_id, option_id)

FOREIGN KEY (view_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (option_id) references SYS.ISYSMVOPTIONNAME (option_id)

See also

- "SYSMVOPTIONNAME system view" on page 774

SYSMVOPTIONNAME system view

Each row in the SYSMVOPTIONNAME system view contains the name of an option defined in the SYSMVOPTION system view. The underlying system table for this view is ISYSMVOPTIONNAME.

Columns

Column name	Column type	Column constraint
option_id	UNSIGNED INT	NOT NULL
option_name	CHAR(128)	NOT NULL

option_id A number uniquely identifying the option in the database.

option_name The name of the option.

Constraints on underlying system table

PRIMARY KEY (option_id)

See also

- "SYSMVOPTION system view" on page 774

SYSOBJECT system view

Each row in the SYSOBJECT system view describes a database object. The underlying system table for this view is ISYSOBJECT.

Columns

Column name	Column type	Column constraint
object_id	UNSIGNED BIGINT	NOT NULL
status	TINYINT	NOT NULL
object_type	TINYINT	NOT NULL
creation_time	TIMESTAMP	NOT NULL

object_id The internal ID for the object, uniquely identifying it in the database.

status The status of the object. Values include:

- **1 (valid)** The object is available for use by the database server. This status is synonymous with ENABLED. That is, if you ENABLE an object, the status changes to VALID.
- **2 (invalid)** An attempt to recompile the object after an internal operation has failed, for example, after a schema-altering modification to an object on which it depends. The database server continues to try to recompile the object whenever it is referenced in a statement.
- **4 (disabled)** The object has been explicitly disabled by the user, for example using an ALTER TABLE...DISABLE VIEW DEPENDENCIES statement.

object_type Type of object. Values include:

Value	Meaning
1	Table
2	View
3	Materialized view
4	Column
5	Index
6	Procedure
7	Trigger
8	Event
9	User
10	Publication
11	Remote type
12	Login mapping
13	JAR
14	Java class
16	Service

creation_time The date and time when the object was created.

Constraints on underlying system table

PRIMARY KEY (object_id)

SYSOPTION system view

The SYSOPTION system view contains the options one row for each option setting stored in the database. Each user can have their own setting for a given option. In addition, settings for the PUBLIC user ID define the default settings to be used for users that do not have their own setting. The underlying system table for this view is ISYSOPTION.

Columns

Column name	Column type	Column constraint
user_id	UNSIGNED INT	NOT NULL
"option"	CHAR(128)	NOT NULL

Column name	Column type	Column constraint
"setting"	LONG VARCHAR	NOT NULL

user_id The user number to whom the option setting applies.

option The name of the option.

setting The current setting for the option.

Constraints on underlying system table

PRIMARY KEY (user_id, "option")

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSOPTSTAT system view

The SYSOPTSTAT system view stores the cost model calibration information as computed by the ALTER DATABASE CALIBRATE statement. The contents of this view are for internal use only and are best accessed via the sa_get_dtt system procedure. The underlying system table for this view is ISYSOPTSTAT.

Columns

Column name	Column type	Column constraint
stat_id	UNSIGNED INT	NOT NULL
group_id	UNSIGNED INT	NOT NULL
format_id	SMALLINT	NOT NULL
data	LONG BINARY	

stat_id For system use only.

group_id For system use only.

format_id For system use only.

data For system use only.

Constraints on underlying system table

PRIMARY KEY (stat_id, group_id, format_id)

SYSPHYSIDX system view

Each row in the SYSPHYSIDX system view defines a physical index in the database. The underlying system table for this view is ISYSPHYSIDX.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
phys_index_id	UNSIGNED INT	NOT NULL
root	INTEGER	NOT NULL
key_value_count	UNSIGNED INT	NOT NULL
leaf_page_count	UNSIGNED INT	NOT NULL
depth	UNSIGNED SMALLINT	NOT NULL
max_key_distance	UNSIGNED INT	NOT NULL
seq_transitions	UNSIGNED INT	NOT NULL
rand_transitions	UNSIGNED INT	NOT NULL
rand_distance	UNSIGNED INT	NOT NULL
allocation_bitmap	LONG VARBIT	
long_value_bitmap	LONG VARBIT	

table_id The object ID of the table to which the index corresponds.

phys_index_id The unique number of the physical index within its table.

root Identifies the location of the root page of the physical index in the database file.

key_value_count The number of distinct key values in the index.

leaf_page_count The number of leaf index pages.

depth The depth (number of levels) of the physical index.

max_key_distance For system use only.

seq_transitions For system use only.

rand_transitions For system use only.

rand_distance For system use only.

allocation_bitmap For system use only.

long_value_bitmap For system use only.

Constraints on underlying system table

PRIMARY KEY (table_id, phys_index_id)

See also

- "SYSIDXCOL system view" on page 770
- "SYSIDX system view" on page 768

SYSPROCEDURE system view

Each row in the SYSPROCEDURE system view describes one procedure in the database. The underlying system table for this view is ISYSPROCEDURE.

Columns

Column name	Column type	Column constraint
proc_id	UNSIGNED INT	NOT NULL
creator	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
proc_name	CHAR(128)	NOT NULL
proc_defn	LONG VARCHAR	
remarks	LONG VARCHAR	
replicate	CHAR(1)	NOT NULL
srvid	UNSIGNED INT	
source	LONG VARCHAR	
avg_num_rows	FLOAT	
avg_cost	FLOAT	
stats	LONG BINARY	

proc_id Each procedure is assigned a unique number (the procedure number).

creator The owner of the procedure.

object_id The internal ID for the procedure, uniquely identifying it in the database.

proc_name The name of the procedure. One creator cannot have two procedures with the same name.

proc_defn The definition of the procedure.

remarks Remarks about the procedure. This value is stored in the ISYSREMARK system table.

replicate (Y/N) Indicates whether the procedure is a primary data source in a Replication Server installation.

srvid If the procedure is a proxy for a procedure on a remote database server, indicates the remote server.

source The preserved source for the procedure. This value is stored in the ISYSSOURCE system table.

avg_num_rows Information collected for use in query optimization when the procedure appears in the FROM clause.

avg_cost Information collected for use in query optimization when the procedure appears in the FROM clause.

stats Information collected for use in query optimization when the procedure appears in the FROM clause.

Constraints on underlying system table

PRIMARY KEY (proc_id)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

UNIQUE (proc_name, creator)

SYSPROCPARM system view

Each row in the SYSPROCPARM system view describes one parameter to a procedure in the database. The underlying system table for this view is ISYSPROCPARM.

Columns

Column name	Column type	Column constraint
proc_id	UNSIGNED INT	NOT NULL
parm_id	SMALLINT	NOT NULL
parm_type	SMALLINT	NOT NULL
parm_mode_in	CHAR(1)	NOT NULL
parm_mode_out	CHAR(1)	NOT NULL
domain_id	SMALLINT	NOT NULL
width	UNSIGNED INT	NOT NULL
scale	SMALLINT	NOT NULL
user_type	SMALLINT	
parm_name	CHAR(128)	NOT NULL
"default"	LONG VARCHAR	

proc_id Uniquely identifies the procedure to which the parameter belongs.

parm_id Each procedure starts numbering parameters at 1. The order of parameter numbers corresponds to the order in which they were defined.

For functions, the first parameter has the name of the function and represents the return value for the function.

parm_type The type of parameter will be one of the following:

- **0** Normal parameter (variable)
- **1** Result variable - used with a procedure that returns result sets
- **2** SQLSTATE error value
- **3** SQLCODE error value
- **4** Return value from function

parm_mode_in (Y/N) Indicates whether the parameter supplies a value to the procedure (IN or INOUT parameters).

parm_mode_out (Y/N) Indicates whether the parameter returns a value from the procedure (OUT or INOUT parameters) or columns in the RESULT clause.

domain_id Identifies the data type for the parameter, by the data type number listed in the SYSDOMAIN system view.

width Contains the length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.

scale For numeric data types, the number of digits after the decimal point. For all other data types, the value of this column is 1.

user_type The user type of the parameter, if applicable.

parm_name The name of the procedure parameter.

"default" Default value of the parameter. Provided for informational purposes only.

Constraints on underlying system table

PRIMARY KEY (proc_id, parm_id)

FOREIGN KEY (proc_id) references SYS.ISYSPROCEDURE (proc_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

FOREIGN KEY (user_type) references SYS.ISYSUSERTYPE (type_id)

SYSPROCPERM system view

Each row of the SYSPROCPERM system view describes a user granted permission to execute a procedure. Only users who have been granted permission can execute a procedure. The underlying system table for this view is ISYSPROCPERM.

Columns

Column name	Column type	Column constraint
proc_id	UNSIGNED INT	NOT NULL
grantee	UNSIGNED INT	NOT NULL

proc_id The procedure number uniquely identifies the procedure for which permission has been granted.

grantee The user number of the user receiving the permission.

Constraints on underlying system table

PRIMARY KEY (proc_id, grantee)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

FOREIGN KEY (proc_id) references SYS.ISYSPROCEDURE (proc_id)

SYSPROXYTAB system view

Each row of the SYSPROXYTAB system view describes the remote parameters of one proxy table. The underlying system table for this view is ISYSPROXYTAB.

Columns

Column name	Column type	Column constraint
table_object_id	UNSIGNED BIGINT	NOT NULL
existing_obj	CHAR(1)	
srvid	UNSIGNED INT	
remote_location	LONG VARCHAR	

table_object_id The object ID of the proxy table.

existing_obj Indicates whether the proxy table previously existed on the remote server (Y/N).

srvid The unique ID for the remote server associated with the proxy table.

remote_location The location of the proxy table on the remote server.

Constraints on underlying system table

PRIMARY KEY (table_object_id)

FOREIGN KEY (table_object_id) references ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

SYSPUBLICATION system view

Each row in the SYSPUBLICATION system view describes a SQL Remote or MobiLink publication. The underlying system table for this view is ISYSPUBLICATION.

Columns

Column name	Column type	Column constraint
publication_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
publication_name	CHAR(128)	NOT NULL
remarks	LONG VARCHAR	
type	CHAR(1)	NOT NULL
sync_type	UNSIGNED INT	NOT NULL

publication_id A number uniquely identifying the publication.

object_id The internal ID for the publication, uniquely identifying it in the database.

creator The owner of the publication.

publication_name The name of the publication.

remarks Remarks about the publication. This value is stored in the ISYSREMARK system table.

type This column is deprecated.

sync_type The type of synchronization for the publication. Values include:

- **logscan** This is a regular publication that uses the transaction log to upload all relevant data that has changed since the last upload.
- **scripted upload** For this publication, the transaction log is ignored and the upload is defined by the user using stored procedures. Information about the stored procedures is stored in the ISYSSYNCSCRIPT system table.
- **download only** This is a download-only publication; no data is uploaded.

Constraints on underlying system table

PRIMARY KEY (publication_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

See also

- "Scripted Upload" [*MobiLink - Client Administration*]
- "SYSSYNCSCRIPT system view" on page 793

SYSREMARK system view

Each row in the SYSREMARK system view describes a remark (or comment) for an object. The underlying system table for this view is ISYSREMARK.

Column

Column	Data type	Column Constraint
object_id	UNSIGNED BIGINT	NOT NULL
remarks	LONG VARCHAR	NOT NULL

object_id The internal ID for the object that has an associated remark.

remarks The remark or comment associated with the object.

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSREMOTEOPTION system view

Each row in the SYSREMOTEOPTION system view describes the value of a SQL Remote message link parameter. The underlying system table for this view is ISYSREMOTEOPTION.

Columns

Column	Data type	Column Constraint
option_id	UNSIGNED INT	NOT NULL
user_id	UNSIGNED INT	NOT NULL
"setting"	VARCHAR(255)	NOT NULL

Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSREMOTEOPTION2 view provides public access to the data in this view except for the potentially sensitive columns.

option_id An identification number for the message link parameter.

user_id The user ID for which the parameter is set.

"setting" The value of the message link parameter.

Constraints on underlying system table

PRIMARY KEY (option_id, user_id)

FOREIGN KEY (option_id) references SYS.ISYSREMOTEOPTIONTYPE (option_id)

SYSREMOTEOPTIONTYPE system view

Each row in the SYSREMOTEOPTIONTYPE system view describes one of the SQL Remote message link parameters. The underlying system table for this view is ISYSREMOTEOPTIONTYPE.

Columns

Column	Data type	Column constraint
option_id	UNSIGNED INT	NOT NULL
type_id	SMALLINT	NOT NULL
"option"	VARCHAR(128)	NOT NULL

option_id An identification number for the message link parameter.

type_id An identification number for the message type that uses the parameter.

"option" The name of the message link parameter.

Constraints on underlying system table

PRIMARY KEY (option_id)

FOREIGN KEY (type_id) references SYS.ISYSREMOTETYPE (type_id)

SYSREMOTETYPE system view

The SYSREMOTETYPE system view contains information about SQL Remote. The underlying system table for this view is ISYSREMOTETYPE.

Columns

Column name	Column type	Column constraint
type_id	SMALLINT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
type_name	CHAR(128)	NOT NULL
publisher_address	LONG VARCHAR	NOT NULL
remarks	LONG VARCHAR	

type_id Identifies which of the message systems supported by SQL Remote is to be used to send messages to the user.

object_id The internal ID for the remote type, uniquely identifying it in the database.

type_name The name of the message system supported by SQL Remote.

publisher_address The address of the remote database publisher.

remarks Remarks about the remote type. This value is stored in the ISYSREMARK system table.

Constraints on underlying system table

PRIMARY KEY (type_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

UNIQUE (type_name)

SYSREMOTEUSER system view

Each row in the SYSREMOTEUSER system view describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user. The underlying system table for this view is ISYSREMOTEUSER.

Columns

Column name	Column type	Column constraint
user_id	UNSIGNED INT	NOT NULL
consolidate	CHAR(1)	NOT NULL
type_id	SMALLINT	NOT NULL
address	LONG VARCHAR	NOT NULL
frequency	CHAR(1)	NOT NULL
send_time	TIME	
log_send	UNSIGNED BIGINT	NOT NULL
time_sent	TIMESTAMP	
log_sent	UNSIGNED BIGINT	NOT NULL
confirm_sent	UNSIGNED BIGINT	NOT NULL
send_count	INTEGER	NOT NULL
resend_count	INTEGER	NOT NULL
time_received	TIMESTAMP	

Column name	Column type	Column constraint
log_received	UNSIGNED BIGINT	NOT NULL
confirm_received	UNSIGNED BIGINT	
receive_count	INTEGER	NOT NULL
rereceive_count	INTEGER	NOT NULL

user_id The user number of the user with REMOTE permissions.

consolidate (Y/N) Indicates whether the user was granted CONSOLIDATE permissions (Y) or REMOTE permissions (N).

type_id Identifies which of the message systems supported by SQL Remote is used to send messages to the user.

address The address to which SQL Remote messages are to be sent. The address must be appropriate for the address_type.

frequency How frequently SQL Remote messages are sent.

send_time The next time messages are to be sent to this user.

log_send Messages are sent only to subscribers for whom log_send is greater than log_sent.

time_sent The time the most recent message was sent to this subscriber.

log_sent The log offset for the most recently sent operation.

confirm_sent The log offset for the most recently confirmed operation from this subscriber.

send_count How many SQL Remote messages have been sent.

resend_count Counter to ensure that messages are applied only once at the subscriber database.

time_received The time when the most recent message was received from this subscriber.

log_received The log offset in the subscriber's database for the operation that was most recently received at the current database.

confirm_received The log offset in the subscriber's database for the most recent operation for which a confirmation message has been sent.

receive_count How many messages have been received.

rereceive_count Counter to ensure that messages are applied only once at the current database.

Constraints on underlying system table

PRIMARY KEY (user_id)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY (type_id) references SYS.ISYSREMOTETYPE (type_id)

SYSSCHEDULE system view

Each row in the SYSSCHEDULE system view describes a time at which an event is to fire, as specified by the SCHEDULE clause of CREATE EVENT. The underlying system table for this view is ISYSSCHEDULE.

Columns

Column name	Column type	Column constraint
event_id	UNSIGNED INT	NOT NULL
sched_name	VARCHAR(128)	NOT NULL
recurring	TINYINT	NOT NULL
start_time	TIME	NOT NULL
stop_time	TIME	
start_date	DATE	
days_of_week	TINYINT	
days_of_month	UNSIGNED INT	
interval_units	CHAR(10)	
interval_amt	INTEGER	

event_ id The unique number assigned to each event.

sched_name The name associated with the schedule for the event.

recurring (0/1) Indicates if the schedule is repeating.

start_time The schedule start time.

stop_time The schedule stop time if BETWEEN was used.

start_date The first date on which the event is scheduled to execute.

days_of_week A bit mask indicating the days of the week on which the event is scheduled:

- x01 = Sunday
- x02 = Monday
- x04 = Tuesday
- x08 = Wednesday
- x10 = Thursday
- x20 = Friday

- x40 = Saturday

days_of_month A bit mask indicating the days of the month on which the event is scheduled. Some examples include:

- x01 = first day
- x02 = second day
- x40000000 = 31st day
- x80000000 = last day of month

interval_units The interval unit specified by EVERY:

- HH = hours
- NN = minutes
- SS = seconds

interval_amt The period specified by EVERY.

Constraints on underlying system table

PRIMARY KEY (event_id, sched_name)

FOREIGN KEY (event_id) references SYS.ISYSEVENT (event_id)

SYSSERVER system view

Each row in the SYSSERVER system view describes a remote server. The underlying system table for this view is ISYSSERVER.

Note
Previous versions of the catalog contained a SYSSERVERS system table. That table has been renamed to be ISYSSERVER (without an 'S'), and is the underlying table for this view.

Columns

Column name	Column type	Column constraint
srvid	UNSIGNED INT	NOT NULL
srvname	VARCHAR(128)	NOT NULL
srvclass	LONG VARCHAR	NOT NULL
srvinfo	LONG VARCHAR	
srvreadonly	CHAR(1)	NOT NULL

srvid An identifier for the remote server.

srvname The name of the remote server.

srvclass The server class, as specified in the CREATE SERVER statement.

srvinfo Server information.

srvreadonly Y if the server is read only, and N otherwise.

Constraints on underlying system table

PRIMARY KEY (srvid)

SYSSOURCE system view

Each row in the SYSSOURCE system view contains the source code, if applicable, for an object listed in the SYSOBJECT system view. The underlying system table for this view is ISYSSOURCE.

Columns

Column name	Column type	Column constraint
object_id	UNSIGNED BIGINT	NOT NULL
source	LONG VARCHAR	NOT NULL

object_id The internal ID for the object whose source code is being defined.

source This column contains the original source code for the object if the preserve_source_format database option is On when the object was created. For more information, see "preserve_source_format option [database]" [*SQL Anywhere Server - Database Administration*].

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSSQLSERVERTYPE system view

The SYSSQLSERVERTYPE system view contains information relating to compatibility with Adaptive Server Enterprise. The underlying system table for this view is ISYSSQLSERVERTYPE.

Columns

Column name	Column type	Column constraint
ss_user_type	SMALLINT	NOT NULL
ss_domain_id	SMALLINT	NOT NULL

Column name	Column type	Column constraint
ss_type_name	VARCHAR (30)	NOT NULL
primary_sa_domain_id	SMALLINT	NOT NULL
primary_sa_user_type	SMALLINT	

ss_user_type The Adaptive Server Enterprise user type.

ss_domain_id The Adaptive Server Enterprise domain id.

ss_type_name The Adaptive Server Enterprise type name.

primary_sa_domain_id The corresponding SQL Anywhere primary domain id.

primary_sa_user_type The corresponding SQL Anywhere primary user type.

Constraints on underlying system table

PRIMARY KEY (ss_user_type)

SYSSUBSCRIPTION system view

Each row in the SYSSUBSCRIPTION system view describes a subscription from one user ID (which must have REMOTE permissions) to one publication. The underlying system table for this view is ISYSSUBSCRIPTION.

Columns

Column name	Column type	Column constraint
publication_id	UNSIGNED INT	NOT NULL
user_id	UNSIGNED INT	NOT NULL
subscribe_by	CHAR(128)	NOT NULL
created	UNSIGNED BIGINT	NOT NULL
started	UNSIGNED BIGINT	

publication_id The identifier for the publication to which the user ID is subscribed.

user_id The ID of the user who is subscribed to the publication.

subscribe_by The value of the SUBSCRIBE BY expression, if any, for the subscription.

created The offset in the transaction log at which the subscription was created.

started The offset in the transaction log at which the subscription was started.

Constraints on underlying system table

PRIMARY KEY (publication_id, user_id, subscribe_by)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSSYNC system view

The SYSSYNC system view contains information relating to MobiLink synchronization. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSSYNC2 view provides public access to the data in this view except for the potentially sensitive columns. The underlying system table for this view is ISYSSYNC.

Columns

Column name	Column type	Column constraint
sync_id	UNSIGNED INT	NOT NULL
type	CHAR(1)	NOT NULL
publication_id	UNSIGNED INT	
progress	UNSIGNED BIGINT	
site_name	CHAR(128)	
"option"	LONG VARCHAR	
server_connect	LONG VARCHAR	
server_conn_type	LONG VARCHAR	
last_download_time	TIMESTAMP	
last_upload_time	TIMESTAMP	NOT NULL
created	UNSIGNED BIGINT	
log_sent	UNSIGNED BIGINT	
generation_number	INTEGER	NOT NULL
extended_state	VARCHAR(1024)	NOT NULL

sync_id A number that uniquely identifies the row.

type The type of synchronization object: D means definition, T means template, and S means site.

publication_id A publication_id found in the SYSPUBLICATION system view.

progress The log offset of the last successful upload.

site_name A MobiLink user name.

"option" Synchronization options.

server_connect The address or URL of the MobiLink server.

server_conn_type The communication protocol, such as TCP/IP, to use when synchronizing.

last_download_time Indicates the last time a download stream was received from the MobiLink server.

last_upload_time Indicates the last time (measured at the MobiLink server) that information was successfully uploaded. The default is jan-1-1990.

created The log offset at which the subscription was created.

log_sent The log progress up to which information has been uploaded. It is not necessary that an acknowledgement of the upload be received for the entry in this column to be updated.

generation_number For file-base downloads, the last generation number received for this subscription. The default is 0.

extended_state Reserved for internal use.

Constraints on underlying system table

PRIMARY KEY (sync_id)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

UNIQUE (publication_id, site_name)

SYSSYNCSCRIPT system view

Each row in the SYSSYNCSCRIPT system view identifies a stored procedure for MobiLink scripted upload. This view is almost identical to the SYSSYNCSCRIPTS view, except that the values in this view are in their raw format. To see them in their human-readable format, see "SYSSYNCSCRIPTS consolidated view" on page 819.

Refer to the "SYSPUBLICATION system view" on page 783 for information about which publications use scripted upload, and refer to the "SYSPROCEDURE system view" on page 779 for the stored procedure definitions.

The underlying system table for this view is ISYSSYNCSCRIPT.

Column name	Column type	Column constraint
pub_object_id	UNSIGNED BIGINT	NOT NULL
table_object_id	UNSIGNED BIGINT	NOT NULL
type	UNSIGNED INT	NOT NULL
proc_object_id	UNSIGNED BIGINT	NOT NULL

Columns

pub_object_id The object ID of the publication to which the script belongs.

table_object_id The object ID of the table to which the script applies.

type The type of upload procedure.

proc_object_id The object ID of the stored procedure to use for the publication.

Constraints on underlying system table

PRIMARY KEY (pub_object_id, table_object_id, type)

FOREIGN KEY (pub_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (table_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (proc_object_id) references SYS.ISYSOBJECT (object_id)

See also

- "Scripted Upload" [*MobiLink - Client Administration*]
- "SYSPUBLICATION system view" on page 783
- "SYSPROCEDURE system view" on page 779
- "SYSSYNCSCRIPTS consolidated view" on page 819

SYSTAB system view

Each row of the SYSTAB system view describes one table or view in the database. Additional information for views can be found in the SYSVIEW system view. The underlying system table for this view is ISYSTAB.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
file_id	SMALLINT	NOT NULL
count	UNSIGNED BIGINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
table_page_count	INTEGER	NOT NULL
ext_page_count	INTEGER	NOT NULL
commit_action	INTEGER	NOT NULL
share_type	INTEGER	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
last_modified_at	TIMESTAMP	NOT NULL

Column name	Column type	Column constraint
table_name	CHAR(128)	NOT NULL
table_type	TINYINT	NOT NULL
replicate	CHAR(1)	NOT NULL
server_type	TINYINT	NOT NULL
tab_page_list	LONG VARBIT	NULL
ext_page_list	LONG VARBIT	NULL
pct_free	UNSIGNED INT	NULL
clustered_index_id	UNSIGNED INT	NULL
encrypted	CHAR(1)	NOT NULL
table_type_str	CHAR(8)	NOT NULL

table_id Each table is assigned a unique number (the table number).

file_id A value indicating which dbspace contains the table.

count The number of rows in the table or materialized view. This value is updated during each successful checkpoint. This number is used by SQL Anywhere when optimizing database access. The count is always 0 for a non-materialized view or remote table.

creator The user number of the owner of the table or view.

table_page_count The total number of main pages used by the underlying table.

ext_page_count The total number of extension pages used by the underlying table.

commit_action For global temporary tables, 0 indicates that the ON COMMIT PRESERVE ROWS clause was specified when the table was created, 1 indicates that the ON COMMIT DELETE ROWS clause was specified when the table was created (the default behavior for temporary tables), and 3 indicates that the NOT TRANSACTIONAL clause was specified when the table was created. For non-temporary tables, commit_action is always 0.

share_type For global temporary tables, 4 indicates that the SHARE BY ALL clause was specified when the table was created, and 5 indicates that the SHARE BY ALL clause was *not* specified when the table was created. For non-temporary tables, share_type is always 5 because the SHARE BY ALL clause cannot be specified when creating non-temporary tables.

object_id The object ID of the table.

last_modified_at The time at which the data in the table was last modified. This column is only updated at checkpoint time.

table_name The name of the table or view. One creator cannot have two tables or views with the same name.

table_type The type of table or view. Values include:

Value	Table type
1	Base table
2	Materialized view
3	Global temporary table
21	View

replicate A value indicating whether the underlying table is a primary data source in a Replication Server installation.

server_type The location of the data for the underlying table. Values include:

Value	Location
1	Local server (SQL Anywhere)
3	Remote server

tab_page_list The set of pages that contain information for the table, expressed as a bitmap. This is for internal use only.

ext_page_list The set of pages that contain row extensions and large object (LOB) pages for the table, expressed as a bitmap. This is for internal use only.

pct_free The PCT_FREE specification for the table, if one has been specified; otherwise, NULL.

clustered_index_id The ID of the clustered index for the table. If none of the indexes are clustered, then this field is NULL.

encrypted A value (Y or N) indicating whether the table or materialized view is encrypted, one of Y or N.

table_type_str Readable value for table_type. Values include:

Value	Table type
BASE	Base table
MAT VIEW	Materialized view
GBL TEMP	Global temporary table
VIEW	View

Constraints on underlying system table

PRIMARY KEY (table_id)

FOREIGN KEY (file_id) references SYS.ISYSFILE (file_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

UNIQUE (table_name, creator)

UNIQUE (table_name)

See also

- "SYSVIEW system view" on page 806

SYSTABCOL system view

The SYSTABCOL system view contains one row for each column of each table and view in the database. The underlying system table for this view is ISYSTABCOL.

Columns

Column name	Column type	Column constraint
table_id	UNSIGNED INT	NOT NULL
column_id	UNSIGNED INT	NOT NULL
domain_id	SMALLINT	NOT NULL
nulls	CHAR(1)	NOT NULL
width	UNSIGNED INT	NOT NULL
scale	SMALLINT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
max_identity	BIGINT	NOT NULL
column_name	CHAR(128)	NOT NULL
"default"	LONG VARCHAR	
user_type	SMALLINT	
column_type	CHAR(1)	NOT NULL
compressed	TINYINT	NOT NULL
collect_stats	TINYINT	NOT NULL
inline_max	SMALLINT	
inline_long	SMALLINT	
lob_index	TINYINT	

table_id The object ID of the table or view to which the column belongs.

column_id The ID of the column. For each table, column numbering starts at 1. This numbering determines the order in which columns are returned by the SELECT command if an ORDER BY clause is not specified:

```
SELECT * FROM table
```

domain_id The data type for the column, indicated by a data type number listed in the SYSDOMAIN system view.

nulls (Y/N) Indicates whether NULL values are allowed in the column.

width The length of a string column, the precision of numeric columns, or the number of bytes of storage for any other data type.

scale The number of digits after the decimal point for NUMERIC or DECIMAL data type columns. For string columns, a value of 1 indicates character-length semantics and 0 indicates byte-length semantics.

object_id The object ID of the table column.

max_identity The largest value of the column, if it is an AUTOINCREMENT, IDENTITY, or GLOBAL AUTOINCREMENT column.

column_name The name of the column.

default The default value for the column. This value, if specified, is only used when an INSERT statement does not specify a value for the column.

user_type The data type, if the column is defined using a user-defined data type.

column_type The type of column (C=computed column, and R=other columns).

compressed Whether this column is stored in a compressed format.

collect_stats Whether the system automatically collects and updates statistics on this column.

inline_max The maximum number of bytes of a BLOB to store in a row. A NULL value indicates that either the default value has been applied, or that the column is not a character or binary type.

A non-NULL inline_max value corresponds to the INLINE value specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the INLINE clause, see "CREATE TABLE statement" on page 450.

inline_long The number of duplicate bytes of a BLOB to store in a row if the BLOB size exceeds the inline_max value. A NULL value indicates that either the default value has been applied, or that the column is not a character or binary type.

A non-NULL inline_long value corresponds to the PREFIX value specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the PREFIX clause, see "CREATE TABLE statement" on page 450.

lob_index Whether to build indexes on BLOB values in the column that exceed an internal threshold size (approximately eight database pages). A NULL value indicates either that the default is applied, or that the

column is not BLOB type. A value of 1 indicates that indexes will be built. A value of 0 indicates that no indexes will be built.

A non-NULL lob_index value corresponds to whether INDEX or NO INDEX was specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the [NO] INDEX clause, see "CREATE TABLE statement" on page 450.

Constraints on underlying system table

PRIMARY KEY (table_id, column_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (user_type) references SYS.ISYSUSERTYPE (type_id)

SYSTABLEPERM system view

Permissions given by the GRANT statement are stored in the SYSTABLEPERM system view. Each row in this view corresponds to one table, one user ID granting the permission (**grantor**) and one user ID granted the permission (**grantee**). The underlying system table for this view is ISYSTABLEPERM.

Columns

Column name	Column type	Column constraint
stable_id	UNSIGNED INT	NOT NULL
grantee	UNSIGNED INT	NOT NULL
grantor	UNSIGNED INT	NOT NULL
selectauth	CHAR(1)	NOT NULL
insertauth	CHAR(1)	NOT NULL
deleteauth	CHAR(1)	NOT NULL
updateauth	CHAR(1)	NOT NULL
updatecols	CHAR(1)	NOT NULL
alterauth	CHAR(1)	NOT NULL
referenceauth	CHAR(1)	NOT NULL

There are several types of permission that can be granted. Each permission can have one of the following three values.

- **N** No, the grantee has not been granted this permission by the grantor.

- **Y** Yes, the grantee has been given this permission by the grantor.
- **G** The grantee has been given this permission and can grant the same permission to another user. See "GRANT statement" on page 548.

Permissions
The grantee might have been given permission for the same table by another grantor. If so, this information would be found in a different row of the SYSTABLEPERM system view.

stable_id The table number of the table or view to which the permissions apply.

grantee The user number of the user ID receiving the permission.

grantor The user number of the user ID granting the permission.

selectauth (Y/N/G) Indicates whether SELECT permission has been granted.

insertauth (Y/N/G) Indicates whether INSERT permission has been granted.

deleteauth (Y/N/G) Indicates whether DELETE permission has been granted.

updateauth (Y/N/G) Indicates whether UPDATE permission has been granted for all columns in the table.

updatecols (Y/N) Indicates whether UPDATE permission has only been granted for some of the columns in the underlying table. If updatecols has the value Y, there will be one or more rows in the SYSCOLPERM system view granting update permission for the columns.

alterauth (Y/N/G) Indicates whether ALTER permission has been granted.

referenceauth (Y/N/G) Indicates whether REFERENCE permission has been granted.

Constraints on underlying system table

PRIMARY KEY (stable_id, grantee, grantor)

FOREIGN KEY (stable_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (ttable_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (grantor) references SYS.ISYSUSER (user_id)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

SYSTRIGGER system view

Each row in the SYSTRIGGER system view describes one trigger in the database. This view also contains triggers that are automatically created for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE). The underlying system table for this view is ISYSTRIGGER.

Columns

Column name	Column type	Column constraint
trigger_id	UNSIGNED INT	NOT NULL
table_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
event	CHAR(1)	NOT NULL
trigger_time	CHAR(1)	NOT NULL
trigger_order	SMALLINT	
foreign_table_id	UNSIGNED INT	
foreign_key_id	UNSIGNED INT	
referential_action	CHAR(1)	
trigger_name	CHAR(128)	
trigger_defn	LONG VARCHAR	NOT NULL
remarks	LONG VARCHAR	
source	LONG VARCHAR	

trigger_id Each trigger is assigned a unique number (the trigger number).

table_id The table ID of the table to which this trigger belongs.

event The event or events that cause the trigger to fire. This single-character value corresponds to the trigger event that was specified when the trigger was created.

- **A** INSERT, DELETE
- **B** INSERT, UPDATE
- **C** UPDATE COLUMNS
- **D** DELETE
- **E** DELETE, UPDATE
- **I** INSERT
- **U** UPDATE
- **M** INSERT, DELETE, UPDATE

trigger_time The time at which the trigger will fire. This single-character value corresponds to the trigger time that was specified when the trigger was created.

- **A** AFTER (row-level trigger)
- **B** BEFORE (row-level trigger)
- **R** RESOLVE
- **S** AFTER (statement-level trigger)

trigger_order The order in which the trigger will fire. This determines the order that triggers are fired when there are triggers of the same type (insert, update, or delete) that fire at the same time (before or after).

foreign_table_id The table number of the table containing a foreign key definition which has a referential triggered action (such as ON DELETE CASCADE).

foreign_key_id The foreign key number of the foreign key for the table referenced by foreign_table_id.

referential_action The action defined by a foreign key. This single-character value corresponds to the action that was specified when the foreign key was created.

- **C** CASCADE
- **D** SET DEFAULT
- **N** SET NULL
- **R** RESTRICT

trigger_name The name of the trigger. One table cannot have two triggers with the same name.

trigger_defn The command that was used to create the trigger.

remarks Remarks about the trigger. This value is stored in the ISYSREMARK system table.

source The SQL source for the trigger. This value is stored in the ISYSSOURCE system table.

Constraints on underlying system table

PRIMARY KEY (trigger_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (foreign_table_id, foreign_key_id) references SYS.ISYSIDX (table_id, index_id)

UNIQUE (table_id, event, trigger_time, trigger_order)

UNIQUE (trigger_name, table_id)

UNIQUE (table_id, foreign_table_id, foreign_key_id, event)

SYSTYPEMAP system view

The SYSTYPEMAP system view contains the compatibility mapping values for entries in the SYSSQLSERVERTYPE system view. The underlying system table for this view is ISYSTYPEMAP.

Columns

Column name	Column type	Column constraint
ss_user_type	SMALLINT	NOT NULL
sa_domain_id	SMALLINT	NOT NULL
sa_user_type	SMALLINT	
nullable	CHAR(1)	

ss_user_type Contains the Adaptive Server Enterprise user type.

sa_domain_id Contains the corresponding SQL Anywhere domain_id.

sa_user_type Contains the corresponding SQL Anywhere user type.

nullable This field describes whether the type allows NULL values.

Constraints on underlying system table

FOREIGN KEY (sa_domain_id) references SYS.ISYSDOMAIN (domain_id)

SYSUSER system view

Each row in the SYSUSER system view describes a user in the system. The underlying system table for this view is ISYSUSER.

Columns

Column name	Column type	Column constraint
user_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
user_name	CHAR(128)	NOT NULL
password	BINARY(128)	

user_id A number uniquely identifying the user in the system.

object_id The internal ID for the user, uniquely identifying the user in the database.

user_name The login name for the user.

password The user's password.

Constraints on underlying system table

PRIMARY KEY (user_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSUSERAUTHORITY system view

Each row of SYSUSERAUTHORITY system view describes an authority granted to one user ID. The underlying system table for this view is ISYSUSERAUTHORITY.

Columns

Column name	Column type	Column constraint
user_id	UNSIGNED INT	NOT NULL
auth	VARCHAR(20)	NOT NULL

user_id The ID of the user to whom the authority belongs.

auth The authority granted to the user.

Constraints on underlying system table

PRIMARY KEY (user_id, auth)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSUSERMESSAGE system view

Each row in the SYSUSERMESSAGE system view holds a user-defined message for an error condition. The underlying system table for this view is ISYSUSERMESSAGE.

> **Note**
> Previous versions of the catalog contained a SYSUSERMESSAGES system table. That table has been renamed to be ISYSUSERMESSAGE (without an 'S'), and is the underlying table for this view.

Columns

Column name	Column type	Column constraint
error	INTEGER	NOT NULL
uid	UNSIGNED INT	NOT NULL
description	VARCHAR(255)	NOT NULL
langid	SMALLINT	NOT NULL

error A unique identifying number for the error condition.

uid The user number that defined the message.

description The message corresponding to the error condition.

langid Reserved.

Constraints on underlying system table

UNIQUE (error, langid)

SYSUSERTYPE system view

Each row in the SYSUSERTYPE system view holds a description of a user-defined data type. The underlying system table for this view is ISYSUSERTYPE.

Columns

Column name	Column type	Column constraint
type_id	SMALLINT	NOT NULL
creator	UNSIGNED INT	NOT NULL
domain_id	SMALLINT	NOT NULL
nulls	CHAR(1)	NOT NULL
width	UNSIGNED INT	NOT NULL
scale	SMALLINT	NOT NULL
type_name	CHAR(128)	NOT NULL
"default"	LONG VARCHAR	
"check"	LONG VARCHAR	

type_id A unique identifying number for the user-defined data type.

creator The user number of the owner of the data type.

domain_id The data type on which this user defined data type is based, indicated by a data type number listed in the SYSDOMAIN system view.

nulls (Y/N/U) Indicates whether the user-defined data type allows nulls. A value of U indicates that nullability is unspecified.

width The length of a string column, the precision of a numeric column, or the number of bytes of storage for any other data type.

scale The number of digits after the decimal point for numeric data type columns, and zero for all other data types.

type_name The name for the data type.

"default" The default value for the data type.

"check" The CHECK condition for the data type.

Constraints on underlying system table

PRIMARY KEY (type_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

UNIQUE (type_name)

SYSVIEW system view

Each row in the SYSVIEW system view describes a view in the database. Additional information about views can also be found in the SYSTAB system view. The underlying system table for this view is ISYSVIEW.

You can also use the sa_materialized_view_info system procedure for a more readable format of the information for materialized views. See "sa_materialized_view_info system procedure" on page 887.

Columns

Column name	Column type	Column constraint
view_object_id	UNSIGNED BIGINT	NOT NULL
view_def	LONG VARCHAR	NOT NULL
mv_build_type	TINYINT	
mv_refresh_type	TINYINT	
mv_use_in_optimization	TINYINT	
mv_last_refreshed_at	TIMESTAMP	
mv_known_stale_at	TIMESTAMP	

view_object_id The object ID of the view.

view_def The definition (query specification) of the view.

mv_build_type Currently unused.

mv_refresh_type Currently unused.

mv_use_in_optimization This column is for materialized views only, and indicates whether the materialized view can be used during query optimization (0=cannot be used in optimization, 1=can be used in optimization). See "Enabling and disabling optimizer use of a materialized view" [*SQL Anywhere Server - SQL Usage*].

mv_last_refreshed_at This column is for materialized views only, and indicates the date and time of the last refresh.

mv_known_stale_at This column is for materialized views only, and indicates the time at which the materialized view became known as stale. This value corresponds to the time at which one of the underlying base tables was detected as having changed. A value of 0 indicates that the view is either fresh, or that it has become stale but the database server has not marked it as such because the view has not been used since it became stale. Use the sa_materialized_view_info system procedure to determine the status of a materialized view. See "sa_materialized_view_info system procedure" on page 887.

Constraints on underlying system table

PRIMARY KEY (view_object_id)

FOREIGN KEY (view_object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

See also

- "SYSTAB system view" on page 794
- "CREATE MATERIALIZED VIEW statement" on page 411
- "REFRESH MATERIALIZED VIEW statement" on page 621
- "CREATE VIEW statement" on page 471

SYSWEBSERVICE system view

Each row in the SYSWEBSERVICE system view holds a description of a web service. The underlying system table for this view is ISYSWEBSERVICE.

Columns

Column name	Column type	Column constraint
service_id	UNSIGNED INT	NOT NULL
object_id	UNSIGNED BIGINT	NOT NULL
service_name	CHAR(128)	NOT NULL
service_type	VARCHAR(40)	NOT NULL
auth_required	CHAR(1)	NOT NULL
secure_required	CHAR(1)	NOT NULL
url_path	CHAR(1)	NOT NULL
user_id	UNSIGNED INT	
parameter	VARCHAR(250)	
statement	LONG VARCHAR	
remarks	LONG VARCHAR	

service_id A unique identifying number for the web service.

object_id The ID of the webservice.

service_name The name assigned to the web service.

service_type The type of the service; for example, RAW, HTTP, XML, SOAP, or DISH.

auth_required (Y/N) Indicates whether all requests must contain a valid user name and password.

secure_required (Y/N) Indicates whether insecure connections, such as HTTP, are to be accepted, or only secure connections, such as HTTPS.

url_path Controls the interpretation of URLs.

user_id If authentication is enabled, identifies the user, or group of users, that have permission to use the service. If authentication is disabled, specifies the account to use when processing requests.

parameter A prefix that identifies the SOAP services to be included in a DISH service.

statement A SQL statement that is always executed in response to a request. If NULL, arbitrary statements contained in each request are executed instead. Ignored for services of type DISH.

remarks Remarks about the webservice. This value is stored in the ISYSREMARK system table.

Constraints on underlying system table

PRIMARY KEY (service_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

UNIQUE (service_name)

Consolidated views

Consolidated views provide data in a form more frequently required by users. For example, consolidated views often provide commonly-needed joins. Consolidated views differ from system views in that they are not just a straight forward view of raw data in a underlying system table(s). For example, many of the columns in the system views are unintelligible ID values, whereas in the consolidated views, they are readable names.

SYSARTICLECOLS consolidated view

Each row in the SYSARTICLECOLS view identifies a column in an article.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSARTICLECOLS"
  as select p.publication_name,t.table_name,c.column_name from
    SYS.SYSARTICLECOL as ac join
    SYS.SYSPUBLICATION as p on p.publication_id = ac.publication_id join
    SYS.SYSTABLE as t on t.table_id = ac.table_id join
    SYS.SYSCOLUMN as c on c.table_id = ac.table_id and
    c.column_id = ac.column_id
```

See also

- "SYSARTICLECOL system view" on page 755
- "SYSPUBLICATION system view" on page 783
- "SYSTABLE compatibility view (deprecated)" on page 828

SYSARTICLES consolidated view

Each row in the SYSARTICLES view describes an article in a publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
CREATE VIEW SYS.SYSARTICLES AS
   SELECT   u1.user_name as publication_owner, p.publication_name,
      u2.user_name as table_owner, t.table_name,
      a.where_expr, a.subscribe_by_expr, a.alias
   FROM  SYS.ISYSARTICLE a
      JOIN SYS.ISYSPUBLICATION p ON ( a.publication_id = p.publication_id )
      JOIN SYS.ISYSTAB t ON ( a.table_id = t.table_id )
      JOIN SYS.ISYSUSER u1 ON ( p.creator = u1.user_id )
      JOIN SYS.ISYSUSER u2 ON ( t.creator = u2.user_id )
```

See also

- "SYSARTICLE system view" on page 754
- "SYSPUBLICATION system view" on page 783
- "SYSTAB system view" on page 794
- "SYSUSER system view" on page 803

SYSCAPABILITIES consolidated view

Each row in the SYSCAPABILITIES view describes a capability. This view gets its data from the ISYSCAPABILITY and ISYSCAPABILITYNAME system tables.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCAPABILITIES"
  as select t1.capid,t1.srvid,t2.capname,t1.capvalue from
    SYS.ISYSCAPABILITY as t1 join
    SYS.ISYSCAPABILITYNAME as t2 on t1.capid = t2.capid
```

See also

- "SYSCAPABILITY system view" on page 755
- "SYSCAPABILITYNAME system view" on page 756

SYSCATALOG consolidated view

Each row in the SYSCATALOG view describes a system table.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCATALOG"( creator,
  tname,dbspacename,tabletype,ncols,primary_key,"check",
  remarks)
  as select up.user_name,tab.table_name,file.dbspace_name,
    if tab.table_type = 'BASE' then 'TABLE' else tab.table_type
    endif,(select count(*) from SYS.SYSCOLUMN where
      SYSCOLUMN.table_id = tab.table_id),
    if tab.primary_root = 0 then 'N' else 'Y' endif,
    if tab.table_type <> 'VIEW' then tab.view_def endif,
    tab.remarks from
    SYS.SYSTABLE as tab key join
    SYS.SYSFILE as file join
    SYS.SYSUSERPERM as up on up.user_id = tab.creator
```

See also

- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSFILE system view" on page 764
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSCOLAUTH consolidated view

Each row in the SYSCOLAUTH view describes the set of privileges (UPDATE, SELECT, or REFERENCES) granted on a column. The SYSCOLAUTH view provides a user-friendly presentation of data in the "SYSCOLPERM system view" on page 757.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.


```
ALTER VIEW "SYS"."SYSCOLAUTH"( grantor,grantee,creator,tname,colname,
  privilege_type,is_grantable)
  as select up1.user_name,up2.user_name,up3.user_name,tab.table_name,
    col.column_name,cp.privilege_type,cp.is_grantable from
    SYS.SYSCOLPERM as cp join
    SYS.SYSUSERPERM as up1 on up1.user_id = cp.grantor join
    SYS.SYSUSERPERM as up2 on up2.user_id = cp.grantee join
    SYS.SYSTABLE as tab on tab.table_id = cp.table_id join
    SYS.SYSUSERPERM as up3 on up3.user_id = tab.creator join
    SYS.SYSCOLUMN as col on col.table_id = cp.table_id and
    col.column_id = cp.column_id
```

See also

- "SYSCOLPERM system view" on page 757
- "SYSUSERPERM compatibility view (deprecated)" on page 830
- "SYSTABLE compatibility view (deprecated)" on page 828

SYSCOLSTATS consolidated view

The SYSCOLSTATS view contains the column statistics that are stored as histograms and used by the optimizer.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLSTATS"
  as select u.user_name,t.table_name,c.column_name,
    s.format_id,s.update_time,s.density,s.max_steps,
    s.actual_steps,s.step_values,s.frequencies from
    SYS.SYSCOLSTAT as s,SYS.SYSTABLE as t,SYS.SYSCOLUMN as c,SYS.SYSUSERPERM
as u where
    s.table_id = c.table_id and
    s.column_id = c.column_id and
    c.table_id = t.table_id and
    t.creator = u.user_id
```

See also

- "SYSCOLSTAT system view" on page 758
- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSCOLUMN compatibility view (deprecated)" on page 825
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSCOLUMNS consolidated view

Each row in the SYSCOLUMNS view describes one column of each table and view in the catalog.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLUMNS"( creator,cname,tname,coltype,nulls,length,
  syslength,in_primary_key,colno,default_value,
  column_kind,remarks)
  as select up.user_name,col.column_name,tab.table_name,dom.domain_name,
```



```
col.nulls,col.width,col.scale,col.pkey,col.column_id,
col."default",col.column_type,col.remarks from
SYS.SYSCOLUMN as col join
SYS.SYSTABLE as tab on(tab.table_id = col.table_id) key join
SYS.SYSDOMAIN as dom join
SYS.SYSUSERPERM as up on up.user_id = tab.creator
```

See also

- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSDOMAIN system view" on page 761
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSFOREIGNKEYS consolidated view

Each row in the SYSFOREIGNKEYS view describes one foreign key for each table in the catalog.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFOREIGNKEYS"( foreign_creator,
  foreign_tname,
  primary_creator,primary_tname,role,columns)
  as select fk_up.user_name,fk_tab.table_name,pk_up.user_name,
    pk_tab.table_name,fk.role,
    (select list(string(fk_col.column_name,' IS ',
      pk_col.column_name)) from
      SYS.SYSFKCOL as fkc join SYS.SYSCOLUMN as fk_col on(
        fkc.foreign_table_id = fk_col.table_id
        and fkc.foreign_column_id = fk_col.column_id),

      SYS.SYSCOLUMN as pk_col where
      fkc.foreign_table_id = fk.foreign_table_id and
      fkc.foreign_key_id = fk.foreign_key_id and
      pk_col.table_id = fk.primary_table_id and
      pk_col.column_id = fkc.primary_column_id) from
    SYS.SYSFOREIGNKEY as fk join
    SYS.SYSTABLE as fk_tab on fk_tab.table_id = fk.foreign_table_id join
    SYS.SYSUSERPERM as fk_up on fk_up.user_id = fk_tab.creator join
    SYS.SYSTABLE as pk_tab on pk_tab.table_id = fk.primary_table_id join
    SYS.SYSUSERPERM as pk_up on pk_up.user_id = pk_tab.creator
```

See also

- "SYSFKCOL compatibility view (deprecated)" on page 825
- "SYSCOLUMN compatibility view (deprecated)" on page 825
- "SYSFOREIGNKEY compatibility view (deprecated)" on page 826
- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSGROUPS consolidated view

There is one row in the SYSGROUPS view for each member of each group. This view describes the many-to-many relationship between groups and members. A group may have many members, and a user may be a member of many groups.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSGROUPS"( group_name,
  member_name)
  as select g.user_name,u.user_name from
    SYS.SYSGROUP,SYS.SYSUSERPERM as g,
    SYS.SYSUSERPERM as u where
    SYSGROUP.group_id = g.user_id
    and SYSGROUP.group_member = u.user_id
```

See also

- "SYSGROUP system view" on page 766
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSINDEXES consolidated view

Each row in the SYSINDEXES view describes one index in the database. As an alternative to this view, you could also use the SYSIDX and SYSIDXCOL system views.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSINDEXES"( icreator,
  iname,fname,creator,tname,indextype,
  colnames,interval,level_num)
  as select up.user_name,idx.index_name,
    file.file_name,up.user_name,
    tab.table_name,
    if idx."unique" = 'N' then 'Non-unique' else
      if idx."unique" = 'U' then 'UNIQUE constraint'
      else 'Unique' endif
    endif,
    (select list(string(c.column_name,
      if ixc."order" = 'A' then ' ASC' else ' DESC' endif)
       order by ixc.table_id asc,ixc.index_id asc,ixc.sequence asc)
      from SYS.ISYSIDXCOL as ixc join SYS.ISYSTABCOL as c on(
      c.table_id = ixc.table_id and
      c.column_id = ixc.column_id) where
      ixc.index_id = idx.index_id and
      ixc.table_id = idx.table_id), 0,0 from
    SYS.SYSTABLE as tab key join
    SYS.SYSFILE as file key join
    SYS.SYSINDEX as idx join
    SYS.SYSUSERPERM as up on up.user_id = idx.creator
```

See also

- "SYSIDXCOL system view" on page 770
- "SYSTABCOL system view" on page 797
- "SYSFILE system view" on page 764
- "SYSINDEX compatibility view (deprecated)" on page 826
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSOPTIONS consolidated view

Each row in the SYSOPTIONS view describes one option created using the SET command. Each user can have their own setting for each option. In addition, settings for the PUBLIC user define the default settings to be used for users that do not have their own setting.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSOPTIONS"( user_name,"option",setting)
  as select up.user_name,opt."option",opt.setting from
    SYS.SYSOPTION as opt key join SYS.SYSUSERPERM as up
```

See also

- "SYSOPTION system view" on page 776
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSPROCAUTH consolidated view

Each row in the SYSPROCAUTH view describes a set of privileges granted on a procedure. As an alternative, you can also use the SYSPROCPERM system view.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPROCAUTH"( grantee,
  creator,procname)
  as select up1.user_name,up2.user_name,p.proc_name from
    SYS.SYSPROCEDURE as p key join
    SYS.SYSPROCPERM as pp join
    SYS.SYSUSERPERM as up1 on up1.user_id = pp.grantee join
    SYS.SYSUSERPERM as up2 on up2.user_id = p.creator
```

See also

- "SYSPROCEDURE system view" on page 779
- "SYSPROCPERM system view" on page 781
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSPROCS consolidated view

The SYSPROCS view shows the procedure or function name, the name of its creator and any comments recorded for the procedure or function.

The tables and columns that make up this view are provided in the ALTER VIEW statement below.

```
ALTER VIEW "SYS"."SYSPROCS"( creator,
  procname,remarks)
  as select u.user_name,p.proc_name,p.remarks from
    SYS.SYSPROCEDURE as p join
    SYS.ISYSUSER as u on u.user_id = p.creator
```


SYSPROCPARMS consolidated view

Each row in the SYSPROCPARMS view describes a parameter to a procedure in the database.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPROCPARMS"( creator,
  procname,parmname,parm_id,parmtype,parmmode,parmdomain,
  length,scale,"default",user_type)
  as select up.user_name,p.proc_name,pp.parm_name,
      pp.parm_id,pp.parm_type,
    if pp.parm_mode_in = 'Y' and pp.parm_mode_out = 'N'
     then 'IN' else
     if pp.parm_mode_in = 'N' and pp.parm_mode_out = 'Y'
      then 'OUT' else 'INOUT'
     endif
    endif,
    dom.domain_name,pp.width,pp.scale,pp."default",
    ut.type_name from
    SYS.SYSPROCPARM as pp join
    SYS.SYSPROCEDURE as p on p.proc_id = pp.proc_id join
    SYS.ISYSUSER as up on up.user_id = p.creator join
    SYS.SYSDOMAIN as dom on dom.domain_id = pp.domain_id left outer join
    SYS.SYSUSERTYPE as ut on ut.type_id = pp.user_type
```

See also

- "SYSPROCPARM system view" on page 780
- "SYSPROCEDURE system view" on page 779
- "SYSUSER system view" on page 803
- "SYSDOMAIN system view" on page 761
- "SYSUSERTYPE system view" on page 805

SYSPUBLICATIONS consolidated view

Each row in the SYSPUBLICATIONS view describes a SQL Remote or MobiLink publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPUBLICATIONS"
  as select u.user_name as creator,
    p.publication_name,
    p.remarks,
    p.type,
    case p.sync_type
    when 0 then 'logscan'
    when 1 then 'scripted upload'
    when 2 then 'download only' else 'invalid'
    end as sync_type from
    SYS.SYSPUBLICATION as p join
    SYS.SYSUSERPERM as u on u.user_id = p.creator
```

See also

- "SYSPUBLICATION system view" on page 783

- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSREMOTEOPTION2 consolidated view

Presents, in a more readable format, the columns from SYSREMOTEOPTION and SYSREMOTEOPTIONTYPE that do not contain sensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEOPTION2"
  as select ISYSREMOTEOPTION.option_id,
    ISYSREMOTEOPTION.user_id,
    SYS.HIDE_FROM_NON_DBA(ISYSREMOTEOPTION.setting) as setting from
    SYS.ISYSREMOTEOPTION
```

See also

- "SYSREMOTEOPTION system view" on page 784
- "SYSREMOTEOPTIONTYPE system view" on page 785

SYSREMOTEOPTIONS consolidated view

Each row of the SYSREMOTEOPTIONS view describes the values of a SQL Remote message link parameter. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSREMOTEOPTION2 view provides public access to the insensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEOPTIONS"
  as select srt.type_name,
    sup.user_name,
    srot."option",
    SYS.HIDE_FROM_NON_DBA(sro.setting) as setting from
    SYS.ISYSREMOTETYPE as srt,
    SYS.ISYSREMOTEOPTIONTYPE as srot,
    SYS.ISYSREMOTEOPTION as sro,
    SYS.ISYSUSER as sup where
    srt.type_id = srot.type_id and
    srot.option_id = sro.option_id and
    sro.user_id = sup.user_id
```

See also

- "SYSREMOTETYPE system view" on page 785
- "SYSREMOTEOPTIONTYPE system view" on page 785
- "SYSREMOTEOPTION system view" on page 784
- "SYSUSER system view" on page 803

SYSREMOTETYPES consolidated view

Each row of the SYSREMOTETYPES view describes one of the SQL Remote message types, including the publisher address.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTETYPES"
  as select SYSREMOTETYPE.type_id,SYSREMOTETYPE.type_name,
     SYSREMOTETYPE.publisher_address,SYSREMOTETYPE.remarks
     from SYS.SYSREMOTETYPE
```

See also

- "SYSREMOTETYPE system view" on page 785

SYSREMOTEUSERS consolidated view

Each row of the SYSREMOTEUSERS view describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEUSERS"
  as select u.user_name,r.consolidate,t.type_name,
    r.address,r.frequency, r.send_time,
    (if r.frequency = 'A' then null else if
      r.frequency = 'P' then
      if r.time_sent is null then current timestamp
       else(select min(minutes(a.time_sent,
         60*hour(a.send_time)+ minute(seconds(a.send_time,59)))) from
         SYS.SYSREMOTEUSER as a where a.frequency = 'P' and
         a.send_time = r.send_time)
       endif else if current date+r.send_time >
        coalesce(r.time_sent,current timestamp) then
          current date+r.send_time else current date+r.send_time+1 endif
      endif endif) as next_send,
    r.log_send,r.time_sent,r.log_sent,r.confirm_sent,r.send_count,
    r.resend_count,r.time_received,r.log_received,
    r.confirm_received,r.receive_count,r.rereceive_count from
    SYS.SYSREMOTEUSER as r key join
    SYS.SYSUSERPERM as u key join
    SYS.SYSREMOTETYPE as t
```

See also

- "SYSREMOTEUSER system view" on page 786
- "SYSUSERPERM compatibility view (deprecated)" on page 830
- "SYSREMOTETYPE system view" on page 785

SYSSUBSCRIPTIONS consolidated view

Each row describes a subscription from one user ID (which must have REMOTE permissions) to one publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSUBSCRIPTIONS"
  as select p.publication_name,u.user_name,s.subscribe_by,
    s.created,s.started from
    SYS.SYSSUBSCRIPTION as s key join
    SYS.SYSPUBLICATION as p join
    SYS.SYSUSERPERM as u on u.user_id = s.user_id
```

See also

- "SYSSUBSCRIPTION system view" on page 791
- "SYSPUBLICATION system view" on page 783
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSSYNC2 consolidated view

The SYSSYNC2 view provides public access to the data found in the SYSSYNC system view—information relating to MobiLink synchronization—without exposing potentially sensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNC2"
  as select ISYSSYNC.sync_id,
    ISYSSYNC.type,
    ISYSSYNC.publication_id,
    ISYSSYNC.progress,
    ISYSSYNC.site_name,
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC."option")
      as "option",
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC.server_connect)
      as server_connect,
    ISYSSYNC.server_conn_type,
    ISYSSYNC.last_download_time,
    ISYSSYNC.last_upload_time,
    ISYSSYNC.created,
    ISYSSYNC.log_sent,
    ISYSSYNC.generation_number,
    ISYSSYNC.extended_state from
    SYS.ISYSSYNC
```

See also

- "SYSSYNC system view" on page 792

SYSSYNCPUBLICATIONDEFAULTS consolidated view

The SYSSYNCPUBLICATIONDEFAULTS view provides the default synchronization settings associated with publications involved in MobiLink synchronization.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCPUBLICATIONDEFAULTS"
  as select s.sync_id,
    p.publication_name,
    SYS.HIDE_FROM_NON_DBA(s."option") as "option",
    SYS.HIDE_FROM_NON_DBA(s.server_connect)
      as server_connect,
    s.server_conn_type from
    SYS.ISYSSYNC as s key join SYS.ISYSPUBLICATION
     as p where s.site_name is null
```

See also

- "SYSSYNC system view" on page 792
- "SYSPUBLICATION system view" on page 783

SYSSYNCS consolidated view

The SYSSYNCS view contains information relating to MobiLink synchronization. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCS"
  as select p.publication_name,s.progress,s.site_name,
    SYS.HIDE_FROM_NON_DBA(s."option") as "option",
    SYS.HIDE_FROM_NON_DBA(s.server_connect) as server_connect,
    s.server_conn_type,s.last_download_time,
    s.last_upload_time,s.created,s.log_sent,s.generation_number,
    s.extended_state from
    SYS.ISYSSYNC as s left outer join
    SYS.ISYSPUBLICATION as p on
    p.publication_id = s.publication_id
```

See also

- "SYSSYNC system view" on page 792
- "SYSPUBLICATION system view" on page 783

SYSSYNCSCRIPTS consolidated view

Each row in the SYSSYNCSCRIPTS view identifies a stored procedure for MobiLink scripted upload. This view is almost identical to the SYSSYNCSCRIPT system view, except that the values are in human-readable format, as opposed to raw data.


```
ALTER VIEW "SYS"."SYSSYNCSCRIPTS"
  as select p.publication_name,
    t.table_name,
    case s.type
    when 0 then 'upload insert'
    when 1 then 'upload delete'
    when 2 then 'upload update' else 'unknown'
    end as type,
    c.proc_name from
    SYS.ISYSSYNCSCRIPT as s join
    SYS.ISYSPUBLICATION as p on p.object_id = s.pub_object_id join
    SYS.ISYSTAB as t on t.object_id = s.table_object_id join
    SYS.ISYSPROCEDURE as c on c.object_id = s.proc_object_id
```

See also

- "Scripted Upload" [*MobiLink - Client Administration*]
- "SYSPUBLICATION system view" on page 783
- "SYSPROCEDURE system view" on page 779
- "SYSSYNCSCRIPT system view" on page 793

SYSSYNCSUBSCRIPTIONS consolidated view

The SYSSYNCSUBSCRIPTIONS view contains the synchronization settings associated with MobiLink synchronization subscriptions.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCSUBSCRIPTIONS"
  as select s.sync_id,
    p.publication_name,
    s.progress,
    s.site_name,
    SYS.HIDE_FROM_NON_DBA(s."option") as "option",
    SYS.HIDE_FROM_NON_DBA(s.server_connect) as server_connect,
    s.server_conn_type,
    s.last_download_time,
    s.last_upload_time,
    s.created,
    s.log_sent,
    s.generation_number,
    s.extended_state from
    SYS.ISYSSYNC as s key join SYS.ISYSPUBLICATION
     as p where
    s.publication_id is not null and
    s.site_name is not null and
    exists(select 1 from SYS.SYSSYNCUSERS as u where
      s.site_name = u.site_name)
```

See also

- "SYSSYNC system view" on page 792
- "SYSPUBLICATION system view" on page 783
- "SYSSYNCUSERS consolidated view" on page 821

SYSSYNCUSERS consolidated view

A view of synchronization settings associated with MobiLink synchronization users.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCUSERS"
  as select ISYSSYNC.sync_id,
    ISYSSYNC.site_name,
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC."option") as "option",
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC.server_connect)
     as server_connect,
    ISYSSYNC.server_conn_type from
    SYS.ISYSSYNC where
    ISYSSYNC.publication_id is null
```

See also

- "SYSSYNC system view" on page 792

SYSTABAUTH consolidated view

The SYSTABAUTH view contains information from the SYSTABLEPERM system view, but in a more readable format.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTABAUTH"( grantor,
  grantee,screator,stname,tcreator,ttname,
  selectauth,insertauth,deleteauth,
  updateauth,updatecols,alterauth,referenceauth)
  as select up1.user_name,up2.user_name,up3.user_name,tab1.table_name,
    up4.user_name,tab2.table_name,tp.selectauth,tp.insertauth,
    tp.deleteauth,tp.updateauth,tp.updatecols,tp.alterauth,
    tp.referenceauth from
    SYS.SYSTABLEPERM as tp join
    SYS.SYSUSERPERM as up1 on up1.user_id = tp.grantor join
    SYS.SYSUSERPERM as up2 on up2.user_id = tp.grantee join
    SYS.SYSTABLE as tab1 on tab1.table_id = tp.stable_id join
    SYS.SYSUSERPERM as up3 on up3.user_id = tab1.creator join
    SYS.SYSTABLE as tab2 on tab2.table_id = tp.ttable_id join
    SYS.SYSUSERPERM as up4 on up4.user_id = tab2.creator
```

See also

- "SYSTABLEPERM system view" on page 799
- "SYSUSERPERM compatibility view (deprecated)" on page 830
- "SYSTABLE compatibility view (deprecated)" on page 828

SYSTRIGGERS consolidated view

Each row in the SYSTRIGGERS view describes one trigger in the database. This view also contains triggers that are automatically created for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE).

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTRIGGERS"( owner,
  trigname,tname,event,trigtime,trigdefn)
  as select up.user_name,trig.trigger_name,tab.table_name,
    if trig.event = 'I' then 'INSERT' else
      if trig.event = 'U' then 'UPDATE' else
        if trig.event = 'C' then 'UPDATE' else
          if trig.event = 'D' then 'DELETE' else
            if trig.event = 'A' then 'INSERT,DELETE' else
              if trig.event = 'B' then 'INSERT,UPDATE' else
                if trig.event = 'E' then 'DELETE,UPDATE' else
                 'INSERT,DELETE,UPDATE' endif
              endif
            endif
          endif
        endif
      endif
    endif,if trig.trigger_time = 'B' or trig.trigger_time = 'P'
      then 'BEFORE' else
      if trig.trigger_time = 'A' or trig.trigger_time = 'S'
      then 'AFTER' else
        if trig.trigger_time = 'R' then 'RESOLVE' else
        'INSTEAD OF' endif
      endif
    endif,trig.trigger_defn from
    SYS.ISYSTRIGGER as trig key join
    SYS.ISYSTAB as tab join
    SYS.SYSUSERPERM as up on up.user_id = tab.creator where
    trig.foreign_table_id is null
```

See also

- "SYSTRIGGER system view" on page 800
- "SYSTAB system view" on page 794
- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSUSEROPTIONS consolidated view

The SYSUSEROPTIONS view contains the option settings that are in effect for each user. If a user has no setting for an option, this view displays the public setting for the option.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSEROPTIONS"( user_name,
  "option",setting)
  as select u.name,
    o."option",
    isnull((select s.setting from
```



```
      SYS.SYSOPTIONS as s where
      s.user_name = u.name and
      s."option" = o."option"),
    o.setting) from
    SYS.SYSOPTIONS as o,SYS.SYSUSERAUTH as u where
    o.user_name = 'PUBLIC'
```

See also

- "SYSOPTIONS consolidated view" on page 814
- "SYSUSERAUTH compatibility view (deprecated)" on page 829

SYSVIEWS consolidated view

Each row of the SYSVIEWS view describes one view, including its view definition.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSVIEWS"( vcreator,
  viewname,viewtext)
  as select u.user_name,t.table_name,t.view_def from
    SYS.SYSTABLE as t join
    SYS.ISYSUSER as u on(u.user_id = t.creator) where
    t.table_type = 'VIEW'
```

Column name	Column type	Column constraint
vcreator	CHAR (128)	NOT NULL
viewname	CHAR(128)	NOT NULL
viewtext	LONG VARCHAR	

See also

- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSUSER system view" on page 803

Compatibility views

Compatibility views are views that are provided for compatibility with pre-10.0.0 versions of SQL Anywhere. Where possible you should use system and consolidated views instead, as support may diminish for some compatibility views in future releases.

SYSCOLLATION compatibility view (deprecated)

The SYSCOLLATION compatibility view contains the collation sequence information for the database. It is obtainable via built-in functions and is not kept in the catalog. Following is definition for this view:

```
ALTER VIEW "SYS"."SYSCOLLATION"
  as select 1 as collation_id,
    DB_PROPERTY('Collation') as collation_label,
    DB_EXTENDED_PROPERTY('Collation','Description')
      as collation_name,
    cast(DB_EXTENDED_PROPERTY('Collation','LegacyData')
      as binary(1280)) as collation_order
```

See also

- "Database-level properties" [*SQL Anywhere Server - Database Administration*]
- "DB_PROPERTY function [System]" on page 147
- "DB_EXTENDED_PROPERTY function [System]" on page 143

SYSCOLLATIONMAPPINGS compatibility view (deprecated)

The SYSCOLLATIONMAPPINGS compatibility view contains only one row with the database collation mapping. It is obtainable via built-in functions and is not kept in the catalog. Following is definition for this view:

```
ALTER VIEW "SYS"."SYSCOLLATIONMAPPINGS"
  as select DB_PROPERTY('Collation') as collation_label,
    DB_EXTENDED_PROPERTY('Collation','Description')
      as collation_name,
    DB_PROPERTY('Charset') as cs_label,
    DB_EXTENDED_PROPERTY('Collation','ASESensitiveSortOrder')
      as so_case_label,
    DB_EXTENDED_PROPERTY('Collation','ASEInsensitiveSortOrder')
      as so_caseless_label,
    DB_EXTENDED_PROPERTY('Charset','java') as jdk_label
```

See also

- "Database-level properties" [*SQL Anywhere Server - Database Administration*]
- "DB_PROPERTY function [System]" on page 147
- "DB_EXTENDED_PROPERTY function [System]" on page 143

SYSCOLUMN compatibility view (deprecated)

The SYSCOLUMN view is provided for compatibility with older versions of SQL Anywhere that offered a SYSCOLUMN system table. However, the previous SYSCOLUMN table has been replaced by the ISYSTABCOL system table, and its corresponding "SYSTABCOL system view" on page 797, which you should use instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLUMN"
  as select b.table_id,
    b.column_id,
    if c.sequence is null then 'N'
    else 'Y'
    endif as pkey,b.domain_id,
    b.nulls,
    b.width,
    b.scale,
    b.object_id,
    b.max_identity,
    b.column_name,
    r.remarks,
    b."default",
    b.user_type,
    b.column_type
    from SYS.ISYSTABCOL as b
    left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
    left outer join SYS.SYSIDXCOL as c on(b.table_id = c.table_id
      and b.column_id = c.column_id and c.index_id = 0)
```

See also

- "SYSTABCOL system view" on page 797
- "SYSREMARK system view" on page 784
- "SYSIDXCOL system view" on page 770

SYSFKCOL compatibility view (deprecated)

Each row of SYSFKCOL describes the association between a foreign column in the foreign table of a relationship and the primary column in the primary table. This view is deprecated; use the SYSIDX and SYSIDXCOL system views instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFKCOL"
  as select a.table_id as foreign_table_id,
    a.index_id as foreign_key_id,
    a.column_id as foreign_column_id,
    a.primary_column_id
    from SYS.SYSIDXCOL as a,
    SYS.SYSIDX as b
    where a.table_id = b.table_id
    and a.index_id = b.index_id
    and b.index_category = 2
```


See also

- "SYSIDX system view" on page 768
- "SYSIDXCOL system view" on page 770

SYSFOREIGNKEY compatibility view (deprecated)

The SYSFOREIGNKEY view is provided for compatibility with older versions of SQL Anywhere that offered a SYSFOREIGNKEY system table. However, the previous SYSFOREIGNKEY system table has been replaced by the ISYSFKEY system table, and its corresponding "SYSFKEY system view" on page 765, which you should use instead.

A foreign key is a relationship between two tables—the foreign table and the primary table. Every foreign key is defined by one row in SYSFOREIGNKEY and one or more rows in SYSFKCOL. SYSFOREIGNKEY contains general information about the foreign key while SYSFKCOL identifies the columns in the foreign key and associates each column in the foreign key with a column in the primary key of the primary table.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFOREIGNKEY"
  as select b.foreign_table_id,
    b.foreign_index_id as foreign_key_id,
    a.object_id,
    b.primary_table_id,
    p.root,
    b.check_on_commit,
    b.nulls,
    a.index_name as role,
    r.remarks,
    b.primary_index_id,
    a.not_enforced as fk_not_enforced,
    10 as hash_limit
    from(SYS.SYSIDX as a left outer join SYS.ISYSPHYSIDX as p
     on(a.table_id = p.table_id and a.phys_index_id = p.phys_index_id))
    left outer join SYS.ISYSREMARK as r on(a.object_id = r.object_id),
     SYS.ISYSFKEY as b
    where a.table_id = b.foreign_table_id
    and a.index_id = b.foreign_index_id
```

See also

- "SYSIDX system view" on page 768
- "SYSPHYSIDX system view" on page 777
- "SYSREMARK system view" on page 784
- "SYSFKEY system view" on page 765

SYSINDEX compatibility view (deprecated)

The SYSINDEX view is provided for compatibility with older versions of SQL Anywhere that offered a SYSINDEX system table. However, the SYSINDEX system table has been replaced by the ISYSIDX system table, and its corresponding "SYSIDX system view" on page 768, which you should use instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSINDEX"
  as select b.table_id,
    b.index_id,
    b.object_id,
    p.root,
    b.file_id,
    case b."unique"
    when 1 then 'Y'
    when 2 then 'U'
    when 3 then 'M'
    when 4 then 'N'
    else 'I'
    end as "unique",
    t.creator,
    b.index_name,
    r.remarks,
    10 as hash_limit
    from(SYS.ISYSIDX as b left outer join SYS.ISYSPHYSIDX as p
      on (b.table_id = p.table_id and b.phys_index_id = p.phys_index_id))
    left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id),
     SYS.SYSTABLE as t
    where t.table_id = b.table_id
    and b.index_category = 3
```

See also

- "SYSIDX system view" on page 768
- "SYSPHYSIDX system view" on page 777
- "SYSTABLE compatibility view (deprecated)" on page 828
- "SYSREMARK system view" on page 784

SYSINFO compatibility view (deprecated)

The SYSINFO view indicates the database characteristics, as defined when the database was created. It always contains only one row. This view is obtainable via built-in functions and is not kept in the catalog. Following is the definition for the SYSINFO view:

```
ALTER VIEW "SYS"."SYSINFO"( page_size,
  encryption,
  blank_padding,
  case_sensitivity,
  default_collation,
  database_version)
  as select db_property('PageSize'),if
    db_property('Encryption') <> 'None' then 'Y'
    else 'N'
    endif,if db_property('BlankPadding') = 'On' then 'Y'
    else 'N'
    endif,if db_property('CaseSensitive') = 'On' then 'Y'
    else 'N'
    endif,db_property('Collation'),
    null
```


See also

- "Database-level properties" [*SQL Anywhere Server - Database Administration*]
- "DB_PROPERTY function [System]" on page 147
- "DB_EXTENDED_PROPERTY function [System]" on page 143

SYSIXCOL compatibility view (deprecated)

The SYSIXCOL view is provided for compatibility with older versions of SQL Anywhere that offered a SYSIXCOL system table. However, the SYSIXCOL system table has been replaced by the ISYSIDXCOL system table, and its corresponding SYSIDXCOL system view. You should switch to using the "SYSIDXCOL system view" on page 770.

Each row of the SYSIXCOL describes a column in an index. The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSIXCOL"
  as select a.table_id,
    a.index_id,
    a.sequence,
    a.column_id,
    a."order"
    from SYS.SYSIDXCOL as a,
     SYS.SYSIDX as b
    where a.table_id = b.table_id
    and a.index_id = b.index_id
    and b.index_category = 3
```

See also

- "SYSIDX system view" on page 768
- "SYSIDXCOL system view" on page 770

SYSTABLE compatibility view (deprecated)

The SYSTABLE view is provided for compatibility with older versions of SQL Anywhere that offered a SYSTABLE system table. However, the SYSTABLE system table has been replaced by the ISYSTAB system table, and its corresponding "SYSTAB system view" on page 794, which you should use instead.

Each row of SYSTABLE view describes one table in the database.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTABLE"
  as select b.table_id,
    b.file_id,
    b.count,
    0 as first_page,
    b.commit_action as last_page,
    COALESCE(ph.root,0) as primary_root,
    b.creator,
    0 as first_ext_page,
```



```
      0 as last_ext_page,
      b.table_page_count,
      b.ext_page_count,
      b.object_id,
      b.table_name,
      case b.table_type
      when 1 then 'BASE'
      when 2 then 'MAT VIEW'
      when 3 then 'GBL TEMP'
      when 4 then 'LCL TEMP'
      when 21 then 'VIEW'
      when 22 then 'JVT'
      else 'INVALID'
      end as table_type,
      v.view_def,
      r.remarks,
      b.replicate,
      p.existing_obj,
      p.remote_location,'T' as remote_objtype,
      p.srvid,
      case b.server_type
      when 1 then 'SA'
      when 2 then 'IQ'
      when 3 then 'OMNI'
      else 'INVALID'
      end as server_type,
      10 as primary_hash_limit,
      0 as page_map_start,
      s.source,
      b."encrypted"
      from SYS.ISYSTAB as b
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
      left outer join SYS.ISYSSOURCE as s on(b.object_id = s.object_id)
      left outer join SYS.ISYSVIEW as v on(b.object_id = v.view_object_id)
      left outer join SYS.ISYSPROXYTAB as p on(b.object_id = p.table_object_id)
      left outer join(SYS.ISYSIDX as i left outer join SYS.ISYSPHYSIDX
        as ph on(i.table_id = ph.table_id and i.phys_index_id =
ph.phys_index_id))
        on(b.table_id = i.table_id and i.index_category = 1 and i.index_id = 0)
```

See also

- "SYSTAB system view" on page 794
- "SYSREMARK system view" on page 784
- "SYSSOURCE system view" on page 790
- "SYSVIEW system view" on page 806
- "SYSPROXYTAB system view" on page 782
- "SYSIDX system view" on page 768
- "SYSPHYSIDX system view" on page 777

SYSUSERAUTH compatibility view (deprecated)

The SYSUSERAUTH view is provided for compatibility with older versions of SQL Anywhere. Use the SYSUSERAUTHORITY system view instead. See "SYSUSERAUTHORITY system view" on page 804.

Each row of the SYSUSERAUTH view describes a user, without exposing their user_id. Instead, each user is identified by their user name. Because this view displays passwords, this view does not have PUBLIC select permission.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERAUTH"( name,
  password,resourceauth,dbaauth,scheduleauth,user_group)
  as select SYSUSERPERM.user_name,SYSUSERPERM.password,
     SYSUSERPERM.resourceauth,SYSUSERPERM.dbaauth,
     SYSUSERPERM.scheduleauth,SYSUSERPERM.user_group
    from SYS.SYSUSERPERM
```

See also

- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSUSERLIST compatibility view (deprecated)

The SYSUSERAUTH view is provided for compatibility with older versions of SQL Anywhere.

Each row of the SYSUSERLIST view describes a user, without exposing their user_id and password. Each user is identified by their user name.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERLIST"( name,
  resourceauth,dbaauth,scheduleauth,user_group)
  as select SYSUSERPERM.user_name,SYSUSERPERM.resourceauth,
     SYSUSERPERM.dbaauth,SYSUSERPERM.scheduleauth,
     SYSUSERPERM.user_group
    from SYS.SYSUSERPERM
```

See also

- "SYSUSERPERM compatibility view (deprecated)" on page 830

SYSUSERPERM compatibility view (deprecated)

This view is deprecated because it only shows the authorities and permissions available in previous versions. You should change your application to use the SYSUSERAUTHORITY system view instead.

Each row of the SYSUSERPERM view describes one user ID.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERPERM"
  as select b.user_id,
    b.object_id,
    b.user_name,
    b.password,
    if exists(select * from SYS.ISYSUSERAUTHORITY where
      ISYSUSERAUTHORITY.user_id = b.user_id
      and ISYSUSERAUTHORITY.auth = 'RESOURCE')
      then 'Y' else 'N' endif as resourceauth,
    if exists(select * from SYS.ISYSUSERAUTHORITY where
      ISYSUSERAUTHORITY.user_id = b.user_id
```



```
  and ISYSUSERAUTHORITY.auth = 'DBA')
  then 'Y' else 'N' endif as dbaauth,'N'
  as scheduleauth,
if exists(select * from SYS.ISYSUSERAUTHORITY where
  ISYSUSERAUTHORITY.user_id = b.user_id
  and ISYSUSERAUTHORITY.auth = 'PUBLISH')
  then 'Y' else 'N' endif as publishauth,
if exists(select * from SYS.ISYSUSERAUTHORITY where
  ISYSUSERAUTHORITY.user_id = b.user_id
  and ISYSUSERAUTHORITY.auth = 'REMOTE DBA')
  then 'Y' else 'N' endif as remotedbaauth,
if exists(select * from SYS.ISYSUSERAUTHORITY where
  ISYSUSERAUTHORITY.user_id = b.user_id
  and ISYSUSERAUTHORITY.auth = 'GROUP')
  then 'Y' else 'N' endif as user_group,
r.remarks from
SYS.ISYSUSER as b left outer join
SYS.ISYSREMARK as r on(b.object_id = r.object_id)
```

See also

- "SYSUSERAUTHORITY system view" on page 804
- "SYSUSER system view" on page 803
- "SYSREMARK system view" on page 784

SYSUSERPERMS compatibility view (deprecated)

This view is deprecated because it only shows the authorities and permissions available in previous versions. You should change your application to use the SYSUSERAUTHORITY system view instead.

Similar to the SYSUSERPERM view, each row of the SYSUSERPERMS view describes one user ID. However, password information is not included. All users are allowed to read from this view.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERPERMS"
  as select SYSUSERPERM.user_id, SYSUSERPERM.user_name,
  SYSUSERPERM.resourceauth, SYSUSERPERM.dbaauth,
  SYSUSERPERM.scheduleauth, SYSUSERPERM.user_group,
  SYSUSERPERM.publishauth, SYSUSERPERM.remotedbaauth,
  SYSUSERPERM.remarks
  from SYS.SYSUSERPERM
```

See also

- "SYSUSERPERM compatibility view (deprecated)" on page 830
- "SYSUSERAUTHORITY system view" on page 804

Views for Transact-SQL compatibility

The Adaptive Server Enterprise and SQL Anywhere system catalogs are different. The Adaptive Server Enterprise system tables and views are owned by the special user **dbo**, and exist partly in the master database, partly in the sybsecurity database, and partly in each individual database. The SQL Anywhere system tables and views are owned by the special user SYS and exist separately in each database.

To assist in preparing compatible applications, SQL Anywhere provides the following set of views owned by the special user dbo, which correspond to their Adaptive Server Enterprise counterparts. Where architectural differences make the contents of a particular Adaptive Server Enterprise table or view meaningless in a SQL Anywhere context, the view is empty, containing just the column names and data types.

View name	Description
syscolumns	One row for each column in a table or view, and for each parameter in a procedure
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving the SQL definition statement
sysindexes	One row for each clustered or nonclustered index, one row for each table with no indexes, and an additional row for each table containing text or image data.
sysobjects	One row for each table, view, procedure, rule, trigger default, log, or (in tempdb only) temporary object
systypes	One row for each system-supplied or user-defined data type
sysusers	One row for each user allowed in the database
syslogins	One row for each valid user account

CHAPTER 7

System Procedures

Contents

Introduction to system procedures 834
System procedures 835
System extended procedures 951
Adaptive Server Enterprise system and catalog procedures 962

Introduction to system procedures

SQL Anywhere includes the following kinds of system procedures:

- Catalog system procedures, for displaying system information in tabular form.
- Extended system procedures for email support and other functions.
- Miscellaneous procedures for controlling server behavior
- Transact-SQL system and catalog procedures. See "Adaptive Server Enterprise system and catalog procedures" on page 962.

See also: "System functions" on page 100.

System procedure definitions

Detailed information about system procedures and functions is available in Sybase Central:

- To view system procedures and functions, right-click a connected database, choose Filter Objects by Owner, and select dbo.
- Open the Procedures & Functions folder for the database.
- You can see the procedure definition by selecting the procedure in the left pane and then clicking the SQL tab in the right pane.

System procedures

System procedures are owned by the user ID dbo. Some of these procedures are for internal system use. This section documents only those not intended solely for system and internal use. You cannot call external functions on Windows CE.

openxml system procedure

Generates a result set from an XML document.

Syntax

openxml(*xml_data*,
xpath [**,** *flags* [**,** *namespaces*]] **)**
WITH (*column-name column-type* [*xpath*],... **)**

Arguments

xml_data The XML on which the result set is based. This can be any string expression, such as a constant, variable, or column.

xpath A string containing an XPath query. XPath allows you to specify patterns that describe the structure of the XML document you are querying. The XPath pattern included in this argument selects the nodes from the XML document. Each node that matches the XPath query in the second *xpath* argument generates one row in the table.

Metaproperties can only be specified in WITH clause *xpath* arguments. A metaproperty is accessed within an XPath query as if it was an attribute. If a *namespaces* is not specified, then by default the prefix mp is bound to the Uniform Resource Identifier (URI) urn:ianywhere-com:sa-xpath-metaprop. If a *namespaces* is specified, this URI must be bound to mp or some other prefix to access metaproperties in the query. Metaproperty names are case sensitive. The openxml statement supports the following metaproperties:

- **@mp:id** returns an ID for a node that is unique within the XML document. The ID for a given node in a given document may change if the database server is restarted. The value of this metaproperty increases with document order.
- **@mp:localname** returns the local part of the node's name, or NULL if the node does not have a name.
- **@mp:prefix** returns the prefix part of the node's name, or NULL if the node does not have a name or if the name is not prefixed.
- **@mp:namespaceuri** returns the URI of the namespace that the node belongs to, or NULL if the node is not in a namespace.
- **@mp:xmltext** returns a subtree of the XML document in XML form. For example, when you match an internal node, you can use this metaproperty to return an XML string, rather than the concatenated values of the descendant text nodes.

flags Indicates the mapping that should be used between the XML data and the result set when an XPath query is not specified in the WITH clause. If the *flags* parameter is not specified, the default behavior is to map attributes to columns in the result set. The *flags* parameter can have one of the following values:

Value	Description
1	XML attributes are mapped to columns in the result set (the default).
2	XML elements are mapped to columns in the result set.

namespace-declaration An XML document. The in-scope namespaces for the query are taken from the root element of the document. If namespaces are specified, then you must include a *flags* argument, even if all the *xpath* arguments are specified.

WITH clause Specifies the schema of the result set and how the value is found for each column in the result set. WITH clause *xpath* arguments are matched relative to the matches for the *xpath* in the second argument. If a WITH clause expression matches more than one node, then only the first node in the document order is used. If the node is not a text node, then the result is found by appending all the text node descendants. If a WITH clause expression does not match any nodes, then the column for that row is NULL.

The openxml WITH clause syntax is similar to the syntax for selecting from a stored procedure.

For information about selecting from a stored procedure, see "FROM clause" on page 535.

column-name The name of the column in the result set.

column-type The data type of the column in the result set. The data type must be compatible with the values selected from the XML document. See "SQL Data Types" on page 47.

Usage

The openxml system procedure parses the *xml_data* and models the result as a tree. The tree contains a separate node for each element, attribute, and text node, or other XML construct. The XPath queries supplied to the openxml system procedure are used to select nodes from the tree, and the selected nodes are then mapped to the result set.

The XML parser used by the openxml system procedure is non-validating, and does not read the external DTD subset or external parameter entities.

When there are multiple matches for a column expression, the first match in the document order (the order of the original XML document before it was parsed) is used. NULL is returned if there are no matching nodes. When an internal node is selected, the result is all the descendant text nodes of the internal node concatenated together.

Columns of type BINARY, LONG BINARY, IMAGE, and VARBINARY are assumed to be in base64-encoded format and are decoded automatically. If you generate XML using the FOR XML clause, these types are base64-encoded, and can be decoded using the openxml system procedure. See "FOR XML and binary data" [*SQL Anywhere Server - SQL Usage*].

The openxml system procedure supports a subset of the XPath syntax, as follows:

- The child, self, attribute, descendant, descendant-or-self, and parent axes are fully supported.
- Both abbreviated and unabbreviated syntax can be used for all supported features. For example, `'a'` is equivalent to `'child::a'` and `'..'` is equivalent to `'parent::node()'`.
- Name tests can use wildcards. For example, `'a/*/b'`.

- The following kind tests are supported: node(), text(), processing-instruction(), and comment().
- Qualifiers of the form *expr1*[*expr2*] and *expr1*[*expr2*="*string*"] can be used, where *expr2* is any supported XPath expression. A qualifier evaluates TRUE if *expr2* matches one or more nodes. For example, `'a [b]'` finds `a` nodes that have at least one `b` child, and `a[b="i"]` finds `a` nodes that have at least one `b` child with a text value of `i`.

See also

- "Using XPath expressions" [*SQL Anywhere Server - SQL Usage*]
- "Importing XML using openxml" [*SQL Anywhere Server - SQL Usage*]
- XPath query language: http://www.w3.org/TR/xpath.

Example

The following query generates a result set from the XML document supplied as the first argument to the openxml system procedure:

```
SELECT * FROM openxml( '<products>
                <ProductType ID="301">Tee Shirt</ProductType>
                <ProductType ID="401">Baseball Cap</ProductType>
                </products>',
                '/products/ProductType' )
WITH ( ProductName LONG VARCHAR 'text()', ProductID CHAR(3) '@ID')
```

This query generates the following result:

ProductName	ProductID
Tee Shirt	301
Baseball Cap	401

In the following example, the first <ProductType> element contains an entity. When you execute the query, this node is parsed as an element with four children: Tee, &, Sweater, and Set. You can use . to concatenate the children together in the result set.

```
SELECT * FROM openxml( '<products>
                <ProductType ID="301">Tee &amp; Sweater Set</ProductType>
                <ProductType ID="401">Baseball Cap</ProductType>
                </products>',
                '/products/ProductType' )
WITH ( ProductName LONG VARCHAR '.', ProductID CHAR(3) '@ID')
```

This query generates the following result:

ProductName	ProductID
Tee Shirt & Sweater Set	301
Baseball Cap	401

The following query uses an equality predicate to generate a result set from the supplied XML document.

```
SELECT * FROM openxml('<EmployeeDirectory>
   <Employee>
```



```
        <column name="EmployeeID">105</column>
        <column name="GivenName">Matthew</column>
        <column name="Surname">Cobb</column>
        <column name="Street">7 Pleasant Street</column>
        <column name="City">Grimsby</column>
        <column name="State">UT</column>
        <column name="PostalCode">02154</column>
        <column name="Phone">6175553840</column>
     </Employee>
    <Employee>
        <column name="EmployeeID">148</column>
        <column name="GivenName">Julie</column>
        <column name="Surname">Jordan</column>
        <column name="Street">1244 Great Plain Avenue</column>
        <column name="City">Woodbridge</column>
        <column name="State">AZ</column>
        <column name="PostalCode">01890</column>
        <column name="Phone">6175557835</column>
     </Employee>
    <Employee>
        <column name="EmployeeID">160</column>
        <column name="GivenName">Robert</column>
        <column name="Surname">Breault</column>
        <column name="Street">358 Cherry Street</column>
        <column name="City">Milton</column>
        <column name="State">PA</column>
        <column name="PostalCode">02186</column>
        <column name="Phone">6175553099</column>
     </Employee>
    <Employee>
        <column name="EmployeeID">243</column>
        <column name="GivenName">Natasha</column>
        <column name="Surname">Shishov</column>
        <column name="Street">151 Milk Street</column>
        <column name="City">Grimsby</column>
        <column name="State">UT</column>
        <column name="PostalCode">02154</column>
        <column name="Phone">6175552755</column>
     </Employee>
  </EmployeeDirectory>', '/EmployeeDirectory/Employee')
  WITH ( EmployeeID INT 'column[@name="EmployeeID"]',
         GivenName    CHAR(20) 'column[@name="GivenName"]',
         Surname      CHAR(20) 'column[@name="Surname"]',
         PhoneNumber  CHAR(10) 'column[@name="Phone"]');
```

This query generates the following result set:

EmployeeID	GivenName	Surname	PhoneNumber
105	Matthew	Cobb	6175553840
148	Julie	Jordan	6175557835
160	Robert	Breault	6175553099
243	Natasha	Shishov	6175552755

The following query uses the XPath @attribute expression to generate a result set:

```
SELECT * FROM openxml( '<Employee
      EmployeeID="105"
```

```
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/Employee' )
WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone')
```

The following query operates on an XML document like the one used in the above query, except that an XML namespace has been introduced. It demonstrates the use of wildcards in the name test for the XPath query, and generates the same result set as the above query.

```
SELECT * FROM openxml( '<Employee  xmlns="http://www.iAnywhere.com/
EmployeeDemo"
      EmployeeID="105"
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/*:Employee' )

WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone')
```

Alternatively, you could specify a namespace declaration:

```
SELECT * FROM openxml( '<Employee  xmlns="http://www.iAnywhere.com/
EmployeeDemo"
      EmployeeID="105"
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/prefix:Employee', 1, '<r xmlns:prefix="http://www.iAnywhere.com/
EmployeeDemo"/>' )
WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone')
```

For more examples of using the openxml system procedure, see “Importing XML using openxml” [*SQL Anywhere Server - SQL Usage*].

sa_ansi_standard_packages system procedure

Returns information about the non-core SQL extensions used in a SQL statement.

Syntax

sa_ansi_standard_packages(*sql-standard-string*, *sql-statement-string* **)**

Arguments

- ***sql-standard-string*** The standard to use for the core extensions. One of SQL:1999 or SQL:2003.
- ***sql-statement-string*** The SQL statement to evaluate.

Remarks

If there are no non-core extensions used for the statement, the result set is empty.

Permissions

None

Side effects

None

See also

- "SQL preprocessor" [*SQL Anywhere Server - Programming*]
- "SQLFLAGGER function [Miscellaneous]" on page 255
- "sql_flagger_error_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "sql_flagger_warning_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Example

Following is an example call to the sa_ansi_standard_packages system procedure:

```
CALL sa_ansi_standard_packages( 'SQL:2003',
'SELECT *
   FROM ( SELECT o.SalesRepresentative,
                 o.Region,
                 SUM( s.Quantity * p.UnitPrice ) AS total_sales,
                 DENSE_RANK() OVER ( PARTITION BY o.Region,
                                     GROUPING( o.SalesRepresentative )
                                     ORDER BY total_sales DESC ) AS
sales_rank
            FROM Product p, SalesOrderItems s, SalesOrders o
            WHERE p.ID = s.ProductID AND s.ID = o.ID
            GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
o.Region ) ) AS DT
   WHERE sales_rank <= 3
   ORDER BY Region, sales_rank')
```

The query generates the following result set:

package_id	package_name
T612	Advanced OLAP operations
T611	Elementary OLAP operations
F591	Derived tables

package_id	package_name
T431	Extended grouping capabilities

sa_audit_string system procedure

Adds a string to the transaction log.

Syntax

sa_audit_string(*string* **)**

Arguments

- ***string*** A string of characters to add to the transaction log.

Remarks

If auditing is turned on, this system procedure adds a comment into the audit log. The string can be a maximum of 200 bytes long.

Permissions

DBA authority required

Side effects

None

See also

- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]
- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]

Example

The following example uses sa_audit_string to add a comment into the audit log:

```
CALL sa_audit_string( 'Auditing test' )
```

sa_check_commit system procedure

Checks for outstanding referential integrity violations before a commit.

Syntax

sa_check_commit(
tname,
keyname
)

Arguments

- ***tname*** A VARCHAR(128) parameter containing the name of a table with a row that is currently violating referential integrity.
- ***keyname*** A VARCHAR(128) parameter containing the name of the corresponding foreign key index.

Remarks

If the database option wait_for_commit is On, or if a foreign key is defined using CHECK ON COMMIT in the CREATE TABLE statement, you can update the database in such a way as to violate referential integrity, as long as these violations are resolved before the changes are committed.

You can use the sa_check_commit system procedure to check whether there are any outstanding referential integrity violations before attempting to commit your changes.

The returned parameters indicate the name of a table containing a row that is currently violating referential integrity, and the name of the corresponding foreign key index.

Permissions

None

Side effects

None

See also

- "wait_for_commit option [database]" [*SQL Anywhere Server - Database Administration*]
- "CREATE TABLE statement" on page 450

Example

The following set of commands can be executed from Interactive SQL. They delete rows from the Departments table in the sample database, in such a way as to violate referential integrity. The call to the sa_check_commit system procedure checks which tables and keys have outstanding violations, and the rollback cancels the change:

```
SET TEMPORARY OPTION wait_for_commit='On'
go
DELETE FROM Departments
go
CREATE VARIABLE tname VARCHAR( 128 );
CREATE VARIABLE keyname VARCHAR( 128 )
go
CALL sa_check_commit( tname, keyname )
go
SELECT tname, keyname
go
ROLLBACK
go
```

sa_clean_database system procedure

Starts the database cleaner and sets the maximum length of time for which it can run.

Syntax

sa_clean_database([*duration*] **)**

Arguments

- ***duration*** The number of seconds that the clean operation is allowed to run. If no argument is specified, or if 0 is specified, the database cleaner runs until all pages in all dbspaces have been cleaned.

Remarks

The database cleaner is an internal task that runs on a default schedule. You can use this system procedure to force the database cleaner to run immediately and to specify how long the cleaner can run each time it is invoked.

Some database tasks, such as processing snapshot isolation transactions, index maintenance, and deleting rows, can execute more efficiently if some portions of the request are deferred to a later time. These deferrable activities typically involve cleanup by removing deleted, historical, and otherwise unnecessary entries from database pages, or reorganizing database pages for more efficient access.

Postponing some of these activities not only allows the current request to finish more quickly, it potentially allows cleanup to occur when the database server is less active. These unnecessary entries are identified so that they are not visible to other transactions; however, they do take up space on a page, and must be removed at some point.

The database cleaner performs any deferred cleanup activities. It is scheduled to run every 20 seconds. When it is invoked, the database cycles sequentially through the database's dbspaces, examining and cleaning each cleanable page before moving on to the next one. When invoked automatically by the database server, the database cleaner is a self-tuning process. The amount of work that the database cleaner performs, and the duration for which it executes, depend on a number of factors, including the fraction of outstanding cleanable pages in a dbspace, the current amount of activity in the database server, and the amount of time that the database cleaner has already spent cleaning. If, after running for 0.5 seconds, the cleaner detects active requests in the server, it stops and reschedules itself to execute at its regular interval. The database cleaner attempts to process pages when there are no other requests executing in the server, and therefore takes advantage of periods of server inactivity.

Database cleaner statistics are available through four database properties:

- **CleanablePagesAdded** returns the number of pages that need to be cleaned
- **CleanablePagesCleaned** returns the number of pages that have already been cleaned
- **CleanableRowsAdded** returns the number of rows that need to be cleaned
- **CleanabledRowsCleaned** returns the number of rows that have already been cleaned

The difference between the values of CleanablePagesAdded and CleanablePagesCleaned indicates how many database pages still require cleaning.

You can use the sa_clean_database system procedure to configure the database cleaner to run until all the pages in a database are cleaned, or to specify a maximum duration for the database cleaner to run.

To further customize the behavior of the database cleaner, you can set up an event that starts the database cleaner if the number of pages or rows that need to be cleaned exceed a specified threshold. See "CREATE EVENT statement" on page 390.

Permissions

DBA authority required

Side effects

None

See also

- "CREATE EVENT statement" on page 390
- CleanablePagesAdded, CleanablePagesCleaned, CleanableRowsAdded, and CleanableRowsCleaned properties: "Database-level properties" [*SQL Anywhere Server - Database Administration*]

Example

The following example sets the duration of the database cleaner to 10 seconds:

```
CALL sa_clean_database( 10 );
```

The following example creates a scheduled event that runs daily to allow the database cleaner to run until all pages in the database are cleaned:

```
CREATE EVENT DailyDatabaseCleanup
SCHEDULE
   START TIME '6:00 pm'
   ON ( 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' )
   HANDLER
      BEGIN
         CALL sa_clean_database( );
      END;
```

The following example forces the database cleaner to run when 20% or more of the pages in the database need to be cleaned:

```
CREATE EVENT PERIODIC_CLEANER
SCHEDULE
BETWEEN '9:00 am' and '5:00 pm'
EVERY 1 HOURS
HANDLER
BEGIN
     DECLARE @num_db_pages INTEGER;
     DECLARE @num_dirty_pages INTEGER;

     -- Get the number of database pages
     SELECT (SUM( DB_EXTENDED_PROPERTY( 'FileSize', t.file_id ) -
                  DB_EXTENDED_PROPERTY( 'FreePages', t.file_id ) ))
     INTO @num_db_pages
     FROM (SELECT file_id FROM SYSFILE) AS t;

     -- Get the number of dirty pages to be cleaned
     SELECT (DB_PROPERTY( 'CleanablePagesAdded' ) -
                  DB_PROPERTY( 'CleanablePagesCleaned' ))
     INTO @num_dirty_pages;

     -- Check whether the number of dirty pages exceeds 20% of
```



```
        -- the size of the database
        IF @num_dirty_pages > @num_db_pages * 0.20 THEN
          -- Start cleaning the database for a maximum of 60 seconds
          CALL sa_clean_database( 60 );
        END IF;
    END
```

sa_column_stats system procedure

Returns various statistics about the specified column(s). These statistics are not related to the column statistics maintained for use by the optimizer.

Syntax

sa_column_stats (
[*tab_name*]
[**,** *col_name*]
[**,** *tab_owner*]
[**,** *max_rows*]
)

Arguments

- ***tab_name*** This optional CHAR(128) parameter specifies the owner of the table. If this parameter is not specified, statistics are calculated for all columns in all table(s).
- ***col_name*** This optional CHAR(128) parameter specifies the columns for which to calculate statistics. If this parameter is not specified, statistics are calculated for all columns in the specified table(s).
- ***tab_owner*** This optional CHAR(128) parameter specifies the owner of the table. If this parameter is not specified, the database server uses the owner of the first table that matches the *tab_name* specified.
- ***max_rows*** This optional INTEGER parameter specifies the number of rows to use for the calculations. If this parameter is not specified, 1000 rows are used by default. Specifying 0 instructs the database server to calculate the ratio based on all of the rows in the table.

Result set

With the exception of table_owner, table_name, and column_name, all values in the result set are NULL for non-string columns. Also, for empty tables, num_rows_processed and num_values_compressed are 0, while all other values are NULL.

Column name	Data type	Description
table_owner	CHAR(128)	The owner of the table.
table_name	CHAR(128)	The table name.
column_name	CHAR(128)	The column name.
num_rows_processed	INTEGER	The total number of rows read to calculate the statistics.
num_values_compressed	INTEGER	The number of values in the column that are compressed. If the column is not compressed, the value is 0.

Column name	Data type	Description
avg_compression_ratio	DOUBLE	The average compression ratio, expressed as a percentage reduction in size, for compressed values in the column. If the column is not compressed, the value is NULL.
avg_length	DOUBLE	The average length of all non-NULL strings in the column.
stddev_length	DOUBLE	The standard deviation of the lengths of all non-NULL strings in the column.
min_length	INTEGER	The minimum length of non-NULL strings in the column.
max_length	INTEGER	The maximum length of strings in the column.
avg_uncompressed_length	DOUBLE	The average length of all uncompressed, non-NULL strings in the column.
stddev_uncompressed_length	DOUBLE	The standard deviation of the lengths of all uncompressed, non-NULL strings in the column.
min_uncompressed_length	INTEGER	The minimum length of all uncompressed, non-NULL strings in the column.
max_uncompressed_length	INTEGER	The maximum length of all uncompressed, non-NULL strings in the column.

Remarks

The database server determines the columns that match the owner, table, and column names specified, and then for each one, calculates statistics for the data in each specified column. By default, the database server only uses the first 1000 rows of data.

For avg_compression_ratio, values cannot be greater than, or equal to 100, however, they can be less than 0 if highly uncompressible data (for example, data that is already compressed) is inserted into a compressed column. Higher values indicate better compression. For example, if the number returned is 80, then the size of the compressed data is 80% less than the size of the uncompressed data.

Permissions

DBA authority required

Side effects

None

See also

- "Choosing whether to compress columns" [*SQL Anywhere Server - SQL Usage*]

Example

In this example, you use the sa_column_stats system procedure in a SELECT statement to determine which columns in the database are benefitting most from column compression:

```
SELECT * FROM sa_column_stats()
  WHERE num_values_compressed > 0
  ORDER BY avg_compression_ratio desc;
```

In this example, you narrow your selection from the previous example to tables owned by bsmith:

```
SELECT * FROM sa_column_stats( tab_owner='bsmith' )
  WHERE num_values_compressed > 0
  ORDER BY avg_compression_ratio desc;
```

sa_conn_activity system procedure

Returns the most recently-prepared SQL statement for each connection to the specified database on the server.

Syntax

sa_conn_activity([*connidparm*] **)**

Arguments

- ***connidparm*** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INT	The ID number of the connection.
Name	VARCHAR(255)	The name of the connection.
Userid	VARCHAR(255)	The user ID for the connection.
DBNumber	INT	The ID number of the database.
LastReqTime	VARCHAR(255)	The time at which the last request for the specified connection started.
LastStatement	LONG VARCHAR	The most recently-prepared SQL statement for the connection.

Remarks

The sa_conn_activity system procedure returns a result set consisting of the most recently-prepared SQL statement for each connection, if the server has been told to collect the information. Recording of statements must be enabled for the database server prior to calling sa_conn_activity. To do this, specify the -zl option when starting the database server, or execute the following:

```
CALL sa_server_option('RememberLastStatement','ON');
```

This procedure is useful when the database server is busy and you want to obtain information about the last SQL statement prepared for each connection. This feature can be used as an alternative to request logging.

For information on the LastStatement property, from which these values are derived, see "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

If *connidparm* is not specified, then information is returned for all connections to all databases running on the database server. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- "Connection-level properties" [*SQL Anywhere Server - Database Administration*]
- "-zl server option" [*SQL Anywhere Server - Database Administration*]
- "sa_server_option system procedure" on page 914

sa_conn_compression_info system procedure

Summarizes communication compression rates.

Syntax

sa_conn_compression_info([*connidparm*] **)**

Arguments

- ***connidparm*** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Type	VARCHAR(20)	A string identifying whether the compression statistics that follow represent either one Connection, or all connections to the Server.
ConnNumber	INTEGER	An INTEGER representing a connection ID. Returns NULL if the Type is Server.
Compression	VARCHAR(10)	A string representing whether or not compression is enabled for the connection. Returns NULL if Type is Server, or ON/OFF if Type is Connection.
TotalBytes	INTEGER	An INTEGER representing the total number of actual bytes both sent and received.
TotalBytesUnComp	INTEGER	An INTEGER representing the number of bytes that would have been sent and received if compression was disabled.
CompRate	NUMERIC(5,2)	A NUMERIC (5,2) representing the overall compression rate. For example, a value of 0 indicates that no compression occurred. A value of 75 indicates that the data was compressed by 75%, or down to one quarter of its original size.

Column name	Data type	Description
CompRateSent	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate for data sent to the client.
CompRateReceived	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate for data received from the client.
TotalPackets	INTEGER	An INTEGER representing the total number of actual packets both sent and received.
TotalPacketsUnComp	INTEGER	An INTEGER representing the total number of packets that would have been sent and received if compression was disabled.
CompPktRate	NUMERIC(5,2)	A NUMERIC (5,2) representing the overall compression rate of packets.
CompPktRateSent	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate of packets sent to the client.
CompPktRateReceived	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate of packets received from the client.

Remarks

If you specify the connection ID number, the sa_conn_compression_info system procedure returns a result set consisting of compression properties for the supplied connection. If no *connection-id* is supplied, this system procedure returns information for all current connections to databases on the server.

For information on the properties these values are derived from, see "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

Permissions

None

Side effects

None

Example

The following example uses the sa_conn_compression_info system procedure to return a result set summarizing compression properties for all connections to the server.

```
CALL sa_conn_compression_info( )
```

Type	ConnNumber	Compression	TotalBytes	...
Connection	79	Off	7841	...
Server	(NULL)	(NULL)	2737761	...
...	...	...	...	...

sa_conn_info system procedure

Reports connection property information.

Syntax

sa_conn_info([*connidparm*] **)**

Arguments

- ***connidparm*** This optional INTEGER parameter specifies the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
Name	VARCHAR (255)	The name of the connection.
Userid	VARCHAR (255)	The user ID for the connection.
DBNumber	INTEGER	The ID number of the database.
LastReqTime	VARCHAR (255)	The time at which the last request for the specified connection started.
ReqType	VARCHAR (255)	A string for the type of the last request.
CommLink	VARCHAR (255)	The communication link for the connection. This is one of the network protocols supported by SQL Anywhere, or local for a same-computer connection.
NodeAddr	VARCHAR (255)	The address of the client in a client/server connection.
ClientPort	INTEGER	The port number on which the client application communicates using TCP/IP.
ServerPort	INTEGER	The port number on which the server communicates using TCP/IP.
BlockedOn	INTEGER	If the current connection is not blocked, this is zero. If it is blocked, the connection number on which the connection is blocked because of a locking conflict.
LockTable	VARCHAR (255)	If the connection is currently waiting for a lock, LockTable will be the name of the table associated with that lock. Otherwise, LockTable will be the empty string.
UncommitOps	INTEGER	The number of uncommitted operations.

Column name	Data type	Description
LockRowID	UNSIGNED BIGINT	If the connection is waiting on a lock that is associated with a particular row identifier, LockRowID contains that row identifier. LockRowID is NULL if the connection is not waiting on a lock associated with a row (that is, it is not waiting on a lock, or it is waiting on a lock that has no associated row).
LockIndexID	INTEGER	If the connection is waiting on a lock that is associated with a particular index, LockIndexID contains the identifier of that index (or -1 if the lock is associated with all indexes on the table in LockTable). LockIndexID is NULL if the connection is not waiting on a lock associated with an index (that is, it is not waiting on a lock, or it is waiting on a lock that has no associated index).

Remarks

If you specify the connection ID number, the sa_conn_info system procedure returns a result set consisting of connection properties for the supplied connection. If no *connidparm* is supplied, this system procedure returns information for all current connections to databases on the server. If *connidparm* is less than zero, option values for the current connection are returned.

In a block situation, the BlockedOn value returned by this procedure allows you to check which users are blocked, and who they are blocked on. The sa_locks system procedure can be used to display the locks held by the blocking connection.

For more information based on any of these properties, you can execute something similar to the following:

```
SELECT *, DB_NAME( DBNumber ),
   CONNECTION_PROPERTY( 'LastStatement', Number )
   FROM sa_conn_info();
```

The value of LockRowID can be used to look up a lock in the output of the sa_locks procedure.

The value in LockIndexID can be used to look up a lock in the output of the sa_locks procedure. Also, the value in LockIndexID corresponds to the primary key of the ISYSIDX system table, which can be viewed using the SYSIDX system view.

Every lock has an associated table, so the value of LockTable can be used to unambiguously determine whether or not a connection is waiting on a lock.

Permissions

None

Side effects

None

See also

- "Connection-level properties" [*SQL Anywhere Server - Database Administration*]
- "sa_locks system procedure" on page 882

- "SYSIDX system view" on page 768

Example

The following example uses the sa_conn_info system procedure to return a result set summarizing connection properties for all connections to the server.

```
CALL sa_conn_info( );
```

Number	Name	Userid	DBNumber	...
79		DBA	0	...
46	Sybase Central 1	DBA	0	...
...	...	...	...	...

sa_conn_list system procedure

Returns a result set containing connection IDs.

Syntax

sa_conn_list(
[*connidparm*]
[**,** *dbidparm*]
)

Arguments

- ***connidparm*** Use this optional INTEGER parameter to specify the ID number of a connection.
- ***dbidparm*** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.

Remarks

If you do not specify any parameters, or if both parameters are NULL, the connection IDs for all connections to all databases running on the database server are returned. If *connidparm* is less than 0, only the connection ID for the current connection is returned. If *connidparm* is NULL and *dbidparm* is less than 0, the connection IDs for just the current database are returned. If *connidparm* is NULL, and *dbidparm* is not NULL and its value is greater than or equal to 0, the connection IDs for only that database are returned.

Permissions

None

Side effects

None

See also

- "sa_db_list system procedure" on page 858
- "sa_conn_options system procedure" on page 853

sa_conn_options system procedure

Returns property information for connection properties that correspond to database options.

Syntax

sa_conn_options([*connidparm*] **)**

Arguments

- ***connidparm*** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
PropNum	INTEGER	The connection property number.
OptionName	VARCHAR(255)	The option name.
OptionDescription	VARCHAR(255)	The option description.
Value	LONG VARCHAR	The option value.

Remarks

Returns the connection ID as Number, and the PropNum, OptionName, OptionDescription, and Value for each available connection property that corresponds to a database option.

If you do not specify *connidparm*, then option values for all connections to the current database are returned. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- "sa_db_list system procedure" on page 858
- "sa_conn_list system procedure" on page 852

sa_conn_properties system procedure

Reports connection property information.

Syntax

sa_conn_properties([*connidparm*] **)**

Arguments

- ***connidparm*** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
PropNum	INTEGER	The connection property number.
PropName	VARCHAR(255)	The connection property name.
PropDescription	VARCHAR(255)	The connection property description.
Value	LONG VARCHAR	The connection property value.

Remarks

Returns the connection ID as Number, and the PropNum, PropName, PropDescription, and Value for each available connection property. Values are returned for all connection properties, database option settings related to connections, and statistics related to connections.

If no *connidparm* is supplied, properties for all connections to the current database are returned. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- “sa_conn_list system procedure” on page 852
- “sa_conn_options system procedure” on page 853
- “System functions” on page 100
- “Connection-level properties” [*SQL Anywhere Server - Database Administration*]

Examples

The following example uses the sa_conn_properties system procedure to return a result set summarizing connection property information for all connections.

```
CALL sa_conn_properties( )
```


Number	PropNum	PropName	...
79	37	CacheHits	...
79	38	CacheRead	...
...	...	...	...

This example uses the sa_conn_properties system procedure to return a list of all connections, in decreasing order by CPU time*:

```
SELECT Number AS connection_number,
    CONNECTION_PROPERTY ( 'Name', Number ) AS connection_name,
    CONNECTION_PROPERTY ( 'Userid', Number ) AS user_id,
  CAST ( Value AS NUMERIC ( 30, 2 ) ) AS approx_cpu_time
  FROM sa_conn_properties()
  WHERE PropName = 'ApproximateCPUTime'
  ORDER BY approx_cpu_time DESC;
```

*Example courtesy of Breck Carter, RisingRoad Professional Services. http://www.risingroad.com

sa_convert_ml_progress_to_timestamp system procedure

For MobiLink scripted uploads only. This converts the progress value for scripted upload from a 64-bit INTEGER to a TIMESTAMP.

Syntax

sa_convert_ml_progress_to_timestamp(*progress* **)**

Arguments

- ***progress*** The function takes one parameter which is an UNSIGNED BIGINT.

Remarks

The function returns the TIMESTAMP that is represented by the value passed in. This procedure is the inverse of sa_convert_timestamp_to_ml_progress.

Permissions

None

Side effects

None

See also

- “sa_convert_timestamp_to_ml_progress system procedure” on page 856
- “Scripted Upload” [*MobiLink - Client Administration*]

Example

```
SELECT sa_convert_timestamp_to_ml_progress( 3600000 );
```


sa_convert_timestamp_to_ml_progress system procedure

For MobiLink scripted uploads only. This converts the progress value for scripted upload from a TIMESTAMP to a 64-bit UNISIGNED BIGINT.

Syntax

sa_convert_timestamp_to_ml_progress([*t1*] **)**

Arguments

- ***t1*** Use this optional TIMESTAMP parameter to specify the progress value to convert to 64-bit UNISIGNED BIGINT.

Remarks

The function returns an UNISIGNED BIGINT that represents the timestamp passed in as a parameter. This procedure is the inverse of sa_convert_ml_progress_to_timestamp.

Permissions

None

Side effects

None

See also

- "sa_convert_ml_progress_to_timestamp system procedure" on page 855
- "Scripted Upload" [*MobiLink - Client Administration*]

Examples

```
SELECT sa_convert_timestamp_to_ml_progress( CURRENT TIMESTAMP );

SELECT sa_convert_timestamp_to_ml_progress( '1900/01/01 1:00' );
```

sa_db_info system procedure

Reports database property information.

Syntax

sa_db_info([*dbidparm*] **)**

Arguments

- ***dbidparm*** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.

Column name	Data type	Description
Alias	VARCHAR(255)	The database name.
File	VARCHAR(255)	The file name of the database root file, including path.
ConnCount	INTEGER	The number of connections to the database.
PageSize	INTEGER	The page size of the database, in bytes.
LogName	VARCHAR(255)	The file name of the transaction log, including path.

Remarks

If you specify a database ID, sa_db_info returns a single row containing the Number, Alias, File, ConnCount, PageSize, and LogName for the specified database.

If no *dbidparm* is supplied, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- "sa_db_properties system procedure" on page 858
- "Database-level properties" [*SQL Anywhere Server - Database Administration*]

Example

The following statement returns a row for each database that is running on the server:

```
CALL sa_db_info( );
```

Property	Value
Number	0
Alias	demo
File	C:\Documents and Settings\All Users\Documents\SQL Anywhere 10\Samples\demo.db
ConnCount	1
PageSize	4096
LogName	C:\Documents and Settings\All Users\Documents\SQL Anywhere 10\Samples\demo.log

sa_db_list system procedure

Returns a database ID.

Syntax

sa_db_list([*dbidparm*] **)**

Arguments

- ***dbidparm*** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the database.

Remarks

If you do not specify a *dbidparm*, or if *dbidparm* is NULL, the IDs for all databases running on the database server are returned. If *dbidparm* is less than 0, then only the ID for the current database is returned.

Permissions

None

Side effects

None

See also

- "sa_conn_list system procedure" on page 852
- "sa_conn_options system procedure" on page 853

sa_db_properties system procedure

Reports database property information.

Syntax

sa_db_properties([*dbidparm*] **)**

Arguments

- ***dbidparm*** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the database.
PropNum	INTEGER	The database property number.

Column name	Data type	Description
PropName	VARCHAR(255)	The database property name.
PropDescription	VARCHAR(255)	The database property description.
Value	LONG VARCHAR	The database property value.

Remarks

If you specify a database ID, the sa_db_properties system procedure returns the database ID number and the PropNum, PropName, PropDescription, and Value for each available database property. Values are returned for all database properties and statistics related to databases.

If *dbidparm* is not specified, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- "sa_db_info system procedure" on page 856
- "Database-level properties" [*SQL Anywhere Server - Database Administration*]

Example

The following example uses the sa_db_properties system procedure to return a result set summarizing database property information for all databases.

```
CALL sa_db_properties( );
```

Number	PropNum	PropName	...
0	0	ConnCount	...
0	1	IdleCheck	...
0	2	IdleWrite	...
...	...	...	...

sa_dependent_views system procedure

Returns the list of all dependent views for a given table or view.

Syntax

sa_dependent_views('*tbl_name* [**,** *owner_name* **')**

Arguments

- ***tbl_name*** Use this CHARACTER parameter to specify the name of the table or view.
- ***owner_name*** Use this optional CHARACTER parameter to specify the owner for *tbl_name*.

Result set

Column name	Data type	Description
table_id	UNSIGNED INTEGER	The object ID of the table or view.
dep_view_id	UNSIGNED INTEGER	The object ID of the dependent views.

Remarks

Use this procedure to obtain the list of IDs of dependent views. Alternatively, you can use the procedure in a statement that returns more information about the views, such as their names.

Permissions

None

Side effects

None

See also

- "SYSDEPENDENCY system view" on page 760
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]

Examples

In this example, the sa_dependent_views system procedure is used to obtain the list of IDs for the views that are dependent on the SalesOrders table. The procedure returns the table_id for SalesOrders, and the dep_view_id for the dependent view, ViewSalesOrders.

```
sa_dependent_views( 'SalesOrders' );
```

In this example, the sa_dependent_views system procedure is used in a SELECT statement to obtain the list of names of views dependent on the SalesOrders table. The procedure returns the ViewSalesOrders view.

```
SELECT t.table_name FROM SYSTAB t,
sa_dependent_views( 'SalesOrders' ) v
WHERE t.table_id = v.dep_view_id;
```

sa_describe_query system procedure

Describes the result set for a query with one row describing each output column of the query.

Syntax

sa_describe_query(
query
[**,** *add_keys*]
)

Arguments

- ***query*** Use this LONG VARCHAR parameter to specify the text of the SQL statement being described.
- ***add_keys*** Use this optional BIT parameter to specify whether to determine a set of columns that uniquely identify rows in the result set for the query being described. The default is 0; the database server does not attempt to identify the columns. See the Remarks section below for a full explanation of this parameter.

Result Set

Column name	Data type	Description
column_number	INTEGER	The ordinal position of the column described by this row, starting at 1.
name	VARCHAR(128)	The name of the column.
domain_id	SMALLINT	The data type of the column. See "SYSDOMAIN system view" on page 761.
domain_name	VARCHAR(128)	The data type name. See "SYSDOMAIN system view" on page 761.
domain_name_with_size	VARCHAR(160)	The data type name, including size and precision (as used in CREATE TABLE or CAST functions).
width	INTEGER	The length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.
scale	INTEGER	The number of digits after the decimal point for numeric data type columns, and zero for all other data types.
declared_width	INTEGER	The length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.
user_type_id	SMALLINT	The type_id of the user-defined data type if there is one, otherwise NULL. See "SYSUSERTYPE system view" on page 805.
user_type_name	VARCHAR(128)	The name of the user-defined data type if there is one, otherwise NULL. See "SYSUSERTYPE system view" on page 805.
correlation_name	VARCHAR(128)	The correlation name associated with the expression if one is available, otherwise NULL.
base_table_id	UNSIGNED INTEGER	The table_id if the expression is a field, otherwise NULL. See "SYSTAB system view" on page 794.

Column name	Data type	Description
base_column_id	UNSIGNED INTEGER	The column_id if the expression is a field, otherwise NULL. See "SYSTABCOL system view" on page 797.
base_owner_name	VARCHAR(128)	The owner name if the expression is a field, otherwise NULL. See "SYSUSER system view" on page 803.
base_table_name	VARCHAR(128)	The table name if the expression is a field, otherwise NULL.
base_column_name	VARCHAR(128)	The column name if the expression is a field, otherwise NULL.
nulls_allowed	BIT	An indicator that is 1 if the expression can be NULL, otherwise 0.
is_autoincrement	BIT	An indicator that is 1 if the expression is a column declared to be autoincrement, otherwise 0.
is_key_column	BIT	An indicator that is 1 if the expression is part of a key for the result set, otherwise 0. See the Remarks section below for more information.
is_added_key_column	BIT	An indicator that is 1 if the expression is an added key column, otherwise 0. See the Remarks section below for more information.

Remarks

The sa_describe_query procedure provides an API-independent mechanism to describe the name and type information for the expressions in the result set of a query.

When 1 is specified for *add_keys*, the sa_describe_query procedure attempts to find a set of columns from the objects being queried that, when combined, can be used as a key to uniquely identify rows in result set of the query being described. The key takes the form of one or more columns from the objects being queried, and may include columns that are not explicitly referenced in the query. If the optimizer finds a key, the column or columns used in the key are identified in the results by an is_key_column value of 1. If no key is found, an error is returned.

For any column that is included in the key but that is not explicitly referenced in the query, the is_added_key_column value is set to 1 to indicate that the column has been added to the results for the procedure; otherwise, the value of is_added_key_column is 0.

If you do not specify *add_keys*, or you specify a value of 0, the optimizer does not attempt to find a key for the result set, and the is_key_column and is_added_key_column columns contain NULL.

The declared_width and width values both describe the size of a column. The declared_width describes the size of the column as defined by the CREATE TABLE statement or by the query, while the width value gives the size of the field when fetched to the client. The client representation of a type may be different from the database server. For example, date and time types are converted to strings if the return_date_time_as_string option is on. For strings, fields declared with character-length semantics have a

declared_width value that matches the CREATE TABLE size, while the width value gives the maximum number of bytes needed to store the returned string. For example:

Declaration	width	declared_width
CHAR(10)	10	10
CHAR(10 CHAR)	40	10
TIMESTAMP	depends on the length of the timestamp format string	8
NUMERIC(10, 3)	10 (precision)	10 (precision)

Permissions

None

Side effects

None

See also

- "EXPRTYPE function [Miscellaneous]" on page 164
- "Character data types" on page 48
- "return_date_time_as_string option [database]" [*SQL Anywhere Server - Database Administration*]

Examples

The following example describes the information returned when querying all columns in the Departments table:

```
SELECT *
FROM sa_describe_query( 'SELECT * FROM Departments DEPT' );
```

The results show the values of the is_key_column and is_added_key_column as NULL because the *add_keys* parameter was not specified.

The following example describes the information returned by querying the DepartmentName and Surname columns of the Employees table, joined with the Departments table:

```
SELECT  *
FROM  sa_describe_query( 'SELECT DepartmentName, Surname
 FROM Employees E JOIN Departments D ON E.EmployeeID = D.DepartmentHeadId',
 add_keys = 1 );
```

The results shows a 1 in rows 3 and 4 of the result set, indicating that the columns needed to uniquely identify rows in the result set for the query are Employees.EmployeeID and Departments.DepartmentID. Also, a 1 is present in the is_added_key_column for rows 3 and 4 because Employees.EmployeeID and Departments.DepartmentID were not explicitly referenced in the query being described.

sa_disable_auditing_type system procedure

Disables auditing of specific events.

Syntax

sa_disable_auditing_type(' *types* **')**

Arguments

- ***types*** Use this VARCHAR(128) parameter to specify a comma-delimited string containing one or more of the following values:

  all disables all types of auditing.

  connect disables auditing of both successful and failed connection attempts.

  connectFailed disables auditing of failed connection attempts.

  DDL disables auditing of DDL statements.

  options disables auditing of public options.

  permission disables auditing of permission checks, user checks, and SETUSER statements.

  permissionDenied disables auditing of failed permission and user checks.

  triggers disables auditing in response to trigger events.

Remarks

You can use the sa_disable_auditing_type system procedure to disable auditing of one or more categories of information.

Setting this option to all disables all auditing. You can also disable auditing by setting the PUBLIC.auditing option to Off.

Permissions

DBA authority required

Side effects

None

See also

- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]
- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]

Example

To disable all auditing:

```
CALL sa_disable_auditing_type( 'all' );
```


sa_disk_free_space system procedure

Reports information about space available for a dbspace, transaction log, transaction log mirror, and/or temporary file.

Syntax

sa_disk_free_space([*p_dbspace_name*] **)**

Arguments

- ***p_dbspace_name*** Use this VARCHAR(128) parameter to specify the name of a dbspace, log file, mirror log file, or temporary file.

  If there is a dbspace called log, mirror, or temp, you can prefix the keyword with an underscore. For example, use _log to get information about the log file if a dbspace called log exists.

  Specify SYSTEM to get information about the main database file, TEMPORARY or TEMP to get information about the temporary file, TRANSLOG to get information about the transaction log, or TRANSLOGMIRROR to get information about the transaction log mirror. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Result set

Column name	Data type	Description
dbspace_name	VARCHAR(128)	This is the dbspace name, transaction log file, mirror log file, or temporary file.
free_space	UNSIGNED BIGINT	The number of free bytes on the volume.

Remarks

If the *p_dbspace_name* parameter is not specified or is NULL, then the result set contains one row for each dbspace, plus one row for each of the transaction log, transaction log mirror, and temporary file, if they exist. If *p_dbspace_name* is specified, then exactly one or zero rows are returned (zero if no such dbspace exists, or if log or mirror is specified and there is no log or mirror file).

For a list of the names of the pre-defined dbspaces for SQL Anywhere databases, see "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Permissions

DBA authority required

Side effects

None

Example

The following example uses the sa_disk_free_space system procedure to return a result set containing information about available space.

```
CALL sa_disk_free_space( )
```


dbspace_name	free_space
SYSTEM	10952101888
Transaction Log	10952101888
Temporary File	10952101888

sa_enable_auditing_type system procedure

Enables auditing and specifies which events to audit.

Syntax

sa_enable_auditing_type(' *types* **')**

Arguments

- ***types*** Use this VARCHAR(128) parameter to specify a comma-delimited string containing one or more of the following values:

  all enables all types of auditing.

  connect enables auditing of both successful and failed connection attempts.

  connectFailed enables auditing of failed connection attempts.

  DDL enables auditing of DDL statements.

  options enables auditing of public options.

  permission enables auditing of permission checks, user checks, and setuser statements.

  permissionDenied enables auditing of failed permission and user checks.

  triggers enables auditing after a trigger event.

Remarks

sa_enable_auditing_type works in conjunction with the PUBLIC.auditing option to enable auditing of specific types of information.

If you set the PUBLIC.auditing option to On, and do not specify which type of information to audit, the default setting (all) takes effect. In this case, all types of auditing information are recorded.

If you set the PUBLIC.auditing option to On, and disable all types of auditing using sa_disable_auditing_type, no auditing information is recorded. To re-establish auditing, you must use sa_enable_auditing_type to specify which type of information you want to audit.

If you set the PUBLIC.auditing option to Off, then no auditing information is recorded, regardless of the sa_enable_auditing_type setting.

Permissions

DBA authority required

Side effects

None

See also

- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]
- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]

Example

To enable only option auditing:

```
CALL sa_enable_auditing_type('options');
```

sa_eng_properties system procedure

Reports database server property information.

Syntax

sa_eng_properties( )

Result set

Column name	Data type	Description
PropNum	INTEGER	The database server property number.
PropName	VARCHAR(255)	The database server property name.
PropDescription	VARCHAR(255)	The database server property description.
Value	LONG VARCHAR	The database server property value.

Remarks

Returns the PropNum, PropName, PropDescription, and Value for each available server property. Values are returned for all database server properties and statistics related to database servers. For a list of available database server properties, see "System functions" on page 100.

Permissions

None

Side effects

None

See also

- "Server-level properties" [*SQL Anywhere Server - Database Administration*]

Example

The following statement returns a set of available server properties


```
CALL sa_eng_properties( );
```

PropNum	PropName	...
1	IdleWrite	…
2	IdleChkPt	…
…	…	…

sa_flush_cache system procedure

Empties all pages for the current database in the database server cache.

Syntax

sa_flush_cache( )

Remarks

Database administrators can use this procedure to empty the contents of the database server cache for the current database. This is useful in performance measurement to ensure repeatable results.

Permissions

DBA authority required

Side effects

None

sa_flush_statistics system procedure

Saves all cost model statistics in the database server cache.

Syntax

sa_flush_statistics( )

Remarks

Use this procedure to flush current cost model statistics in the database, currently cached, to disk. You can then retrieve the statistics using the sa_get_histogram system procedure, or the Histogram utility (dbhist). When this system procedure runs, the ISYSCOLSTAT system table is updated. Under normal operation it should not be necessary to execute this procedure because the server automatically writes out statistics to disk on a periodic basis.

Permissions

DBA authority required

Side effects

None

See also

- "sa_get_histogram system procedure" on page 871
- "SYSCOLSTAT system view" on page 758
- "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*]

sa_get_bits system procedure

Takes a bit string and returns a row for each bit in the string. By default, only rows with a bit value of 1 are returned.

Syntax

sa_get_bits(*bit_string* [**,** *only_on_bits*])

Arguments

- ***bit_string*** Use this LONG VARBIT parameter to specify the bit string from which to get the bits. If the *bit_string* parameter is NULL, no rows are returned.
- ***only_on_bits*** Use this optional BIT to specify whether to return only rows with on bits (bits with the value of 1). Specify 1 (the default) to return only rows with on bits; specify 0 to return rows for all bits in the bit string.

Result set

Column	Data type	Description
bitnum	UNSIGNED INT	The position of the bit described by this row. For example, the first bit in the bit string has bitnum of 1.
bit_val	BIT	The value of the bit at position bitnum. If *only_on_bits* is set to 1, this value is always 1.

Remarks

The sa_get_bits system procedure decodes a bit string, returning one row for each bit in the bit string, indicating the value of the bit. If *only_on_bits* is set to 1 (the default) or NULL, then only rows corresponding to on bits are returned. An optimization allows this case to be processed efficiently for long bit strings that have few on bits. If *only_on_bits* is set to 0, then a row is returned for each bit in the bit string.

For example, the statement `CALL sa_get_bits( '1010' );` returns the following result set, indicating on bits in positions 1 and 3 of the bit string.

bitnum	bit_val
1	1
3	1

The sa_get_bits system procedure can be used to convert a bit string into a relation. This can be used to join a bit string with a table, or to retrieve a bit string as a result set instead of as a single binary value. It can be

more efficient to retrieve a bit string as a result set if there are a large number of 0 bits, as these do not need to be retrieved.

Permissions

None.

Side effects

None.

See also

- "sa_split_list system procedure" on page 927
- "SET_BIT function [Bit array]" on page 244
- "SET_BITS function [Aggregate]" on page 245
- "GET_BIT function [Bit array]" on page 167

Examples

The following example shows how to use the sa_get_bits system procedure to encode a set of integers as a bit string, and then decode it for use in a join:

```
CREATE VARIABLE @s_depts LONG VARBIT;

SELECT  SET_BITS( DepartmentID )
 INTO @s_depts
 FROM Departments
 WHERE DepartmentName like 'S%';

SELECT *
 FROM sa_get_bits( @s_depts ) B
  JOIN Departments D ON B.bitnum = D.DepartmentID;
```

sa_get_dtt system procedure

Reports the current value of the Disk Transfer Time (DTT) model, which is part of the cost model.

Syntax

sa_get_dtt(*file_id* **)**

Arguments

- ***file_id*** Use this UNSIGNED SMALLINT parameter to specify the database file ID.

Remarks

You can obtain the *file_id* from the system table SYSFILE.

This procedure, intended for internal diagnostic purposes, retrieves data from the ISYSOPTSTAT system table.

Result set

Column name	Data type	Description
BandSize	UNSIGNED INTEGER	Size, in pages, of disk over which random access takes place.
ReadTime	UNSIGNED INTEGER	Amortized cost, in microseconds, of reading one page.
WriteTime	UNSIGNED INTEGER	Amortized cost, in microseconds, of writing one page.

Permissions

None

Side effects

None

See also

- "ISYSFILE system table" on page 728
- "SYSOPTSTAT system view" on page 777

sa_get_histogram system procedure

Retrieves the histogram for a column.

Syntax

sa_get_histogram(
col_name,
tbl_name
[**,** *owner_name*]
)

Arguments

- ***col_name*** Use this CHAR(128) parameter to specify the column for which to retrieve the histogram.
- ***tbl_name*** Use this CHAR(128) parameter to specify the table in which *col_name* is found.
- ***owner_name*** Use this optional CHAR(128) parameter to specify the owner of *tbl_name*.

Result set

Column name	Data type	Description
StepNumber	SMALLINT	Histogram bucket number. The frequency of the first bucket (Step-Number = 0) indicates the selectivity of NULLs.
Low	CHAR(128)	Lowest (inclusive) column value in the bucket.
High	CHAR(128)	Highest (exclusive) column value in the bucket.
Frequency	DOUBLE	Selectivity of values in the bucket.

Remarks

This procedure, intended for internal diagnostic purposes, retrieves column statistics from the database server for the specified columns. Note that while these statistics are permanently stored in the system table ISYSCOLSTAT, they are maintained in memory while the server is running, and written to ISYSCOLSTAT periodically. As such, the statistics returned by the sa_get_histogram system procedure may differ from those obtained by selecting from ISYSCOLSTAT at any given point of time.

You can manually update ISYSCOLSTAT with the latest statistics held in memory using the sa_flush_statistics system procedure, however, this is not recommended in a production environment, and should be reserved for diagnostic purposes.

A singleton bucket is indicated by a Low value in the result set being equal to the corresponding High value.

It is recommended that you view histograms using the Histogram utility. See "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*].

To determine the selectivity of a predicate over a string column, you should use the ESTIMATE or ESTIMATE_SOURCE functions. For string columns, both sa_get_histogram and the Histogram utility retrieve nothing from the ISYSCOLSTAT system table. Attempting to retrieve string data generates an error.

Permissions

DBA authority required

Side effects

None

See also

- "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*]
- "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*]
- "ESTIMATE function [Miscellaneous]" on page 156
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 156
- "ISYSCOLSTAT system table" on page 727
- "sa_flush_statistics system procedure" on page 868

sa_get_request_profile system procedure

Analyzes the request log to determine the execution times of similar statements.

Syntax

sa_get_request_profile(
  [*filename*
  [, *conn_id*
  [, *first_file*
  [, *num_files*]]]]
)

Arguments

- ***filename*** Use this optional LONG VARCHAR parameter to specify the request logging file name.

- ***conn_id*** Use this optional UNSIGNED INTEGER parameter to specify the ID number of a connection.
- ***first_file*** Use this optional INTEGER parameter to specify the first request log file to analyze.
- ***num_files*** Use this optional INTEGER parameter to specify the number of request log files to analyze.

Remarks

This procedure calls sa_get_request_times to process a request log file, and then summarizes the results into the global temporary table satmp_request_profile. This table contains the statements from the log along with how many times each was executed, and their total, average, and maximum execution times. The table can be sorted in various ways to identify targets for performance optimization efforts.

If you do not specify a log file (*filename*), the default is the current log file that is specified at the command prompt with -zo, or that has been specified by

```
sa_server_option( 'RequestLogFile', filename )
```

If a connection ID is specified, it is used to filter information from the log so that only requests for that connection are retrieved.

Permissions

DBA authority required

Side effects

Automatic commit

Example

The following command obtains the request times for the requests in the files *req.out.3*, *req.out.4*, and *req.out.5*.

```
CALL sa_get_request_profile('req.out',0,3,3)
```

See also

- "sa_get_request_times system procedure" on page 873
- "sa_statement_text system procedure" on page 929
- "sa_server_option system procedure" on page 914

sa_get_request_times system procedure

Analyzes the request log to determine statement execution times.

Syntax

sa_get_request_times(*filename*
  [, *conn_id*
  [, *first_file*
  [, *num_files*]]]
)

Arguments

- ***filename*** Use this optional LONG VARCHAR parameter to specify the request logging file name.
- ***conn_id*** Use this optional UNSIGNED INTEGER parameter to specify the ID number of a connection.
- ***first_file*** Use this optional INTEGER parameter to specify the first file to analyze.
- ***num_files*** Use this optional INTEGER parameter to specify the number of request log files to analyze.

Remarks

This procedure reads the specified request log and populates the global temporary table satmp_request_time with the statements from the log and their execution times.

For statements such as inserts and updates, the execution time is straightforward. For queries, the time is calculated from preparing the statement to dropping it, including describing it, opening a cursor, fetching rows, and closing the cursor. For most queries, this is an accurate reflection of the amount of time taken. In cases where the cursor is left open while other actions are performed, the time appears as a large value but is not a true indication that the query is costly.

This procedure recognizes host variables in the request log and populates the global temporary table satmp_request_hostvar with their values. For older databases where this temporary table does not exist, host variable values are ignored.

If you do not specify a log file, the default is the current log file that is specified at the command prompt with -zo, or that has been specified by

```
sa_server_option( 'RequestLogFile', filename )
```

If a connection ID is specified, it is used to filter information from the log so that only requests for that connection are retrieved.

Permissions

DBA authority required

Side effects

Automatic commit

Example

The following command obtains the execution times for the requests in the files *req.out.3*, *req.out.4*, and *req.out.5*.

```
CALL sa_get_request_times('req.out',0,3,3)
```

See also

- “sa_get_request_profile system procedure” on page 872
- “sa_statement_text system procedure” on page 929
- “sa_server_option system procedure” on page 914

sa_get_server_messages system procedure

Allows you to return constants from the Server Messages window as a result set.

Syntax

sa_get_server_messages(*first_line* **)**

Arguments

- ***first_line*** Use this INTEGER parameter to specify the line number from which to start displaying server messages.

Result set

Column name	Data type	Description
line_num	INTEGER	The line number of a server message.
message_text	VARCHAR(255)	The server message text.
message_time	TIMESTAMP	The time of the message.

Remarks

This procedure takes an INTEGER parameter that specifies the starting line number to display, and returns a row for that line and for all subsequent lines. If the starting line is negative, the result set starts at the first available line. The result set includes the line number, message text, and message time.

Permissions

None

Side effects

None

Example

The following example uses the sa_get_server_messages system procedure to return a result set containing the content of the Server Messages window starting on line 16.

```
CALL sa_get_server_messages(16);
```

line_num	message_text	...
16	Running on Windows 2000 Build 2195...	...
17	2132K of memory used for caching	...
...	...	...

sa_http_header_info system procedure

Returns HTTP header names and values.

Syntax

sa_http_header_info([*header_parm*] **)**

Arguments

- ***header_parm*** Use this optional VARCHAR(255) parameter to specify an HTTP header name.

Result set

Column name	Data type	Description
Name	VARCHAR(255)	The HTTP header name.
Value	LONG VARCHAR	The HTTP header value.

Remarks

The sa_http_header_info system procedure returns header names and values. If you do not specify the header name using the optional parameter, the result set contains values for all headers.

This procedure returns a non-empty result set if it is called while processing an HTTP request within a web service.

Permissions

None

Side effects

None

See also

- "SQL Anywhere Web Services" [*SQL Anywhere Server - Programming*]
- "sa_http_variable_info system procedure" on page 876

sa_http_variable_info system procedure

Returns HTTP variable names and values.

Syntax

sa_http_variable_info([*variable_parm*] **)**

Arguments

- ***variable_parm*** Use this optional VARCHAR(255) parameter to specify an HTTP variable name.

Result set

Column name	Data type	Description
Name	VARCHAR(255)	The HTTP variable name.
Value	LONG VARCHAR	The HTTP variable value.

Remarks

The sa_http_variable_info system procedure returns variable names and values. If you do not specify the variable name using the optional parameter, the result set contains values for all variables.

This procedure returns a non-empty result set if it is called while processing an HTTP request within a web service.

Permissions

None

Side effects

None

See also

- "SQL Anywhere Web Services" [*SQL Anywhere Server - Programming*]
- "sa_http_header_info system procedure" on page 876

sa_index_density system procedure

Reports information about the amount of fragmentation within database indexes.

Syntax

sa_index_density(
  [*tbl_name*
  [**,** *owner_name*]]
)

Arguments

- ***tbl_name*** Use this optional CHAR(128) parameter to specify the table name.
- ***owner_name*** Use this optional CHAR(128) parameter to specify the owner name.

Result set

Column name	Data type	Description
TableName	CHAR(128)	The name of a table.
TableId	UNSIGNED INTEGER	The table ID.

Column name	Data type	Description
IndexName	CHAR(128)	The name of an index.
IndexId	UNSIGNED INTEGER	The index ID. This column contains one of the following values: ♦ **0** for primary keys ♦ **SYSFKEY.*foreign_key_id*** for foreign keys ♦ **SYSIDX.*index_id*** for all other indexes
IndexType	CHAR(4)	The index type. This column contains one of the following values: ♦ **PKEY** for primary keys ♦ **FKEY** for foreign keys ♦ **UI** for unique indexes ♦ **UC** for unique constraints ♦ **NUI** for non-unique indexes
LeafPages	UNSIGNED INTEGER	The number of leaf pages.
Density	NUMERIC(8,6)	A fraction between 0 and 1 providing an indication of how full each index page is (on average).

Remarks

Database administrators can use this procedure to obtain information about the degree of fragmentation in a database's indexes.

The procedure returns a result set containing the table name, the table ID, the index name, the index ID, the index type, the number of leaf pages, and the index's density.

If you do not specify parameters, the information for all tables appears. Otherwise, the procedure examines only the named table.

For indexes with a high number of leaf pages, higher density values are desirable.

Permissions

DBA authority required

Side effects

None

See also

♦ "Reducing index fragmentation" [*SQL Anywhere Server - SQL Usage*]

Example

The following example uses the sa_index_density system procedure to return a result set summarizing the amount of fragmentation within database indexes.

```
CALL sa_index_density( );
```

TableName	TableId	IndexName	...	Density
Products	436	Products	...	0.012451
...	...	...	...	...

sa_index_levels system procedure

Assists in performance tuning by reporting the number of levels in an index.

Syntax

sa_index_levels(
[*tbl_name*
[**,** *owner_name*]]
)

Arguments

- ***tbl_name*** Use this optional CHAR(128) parameter to specify the table name.
- ***owner_name*** Use this optional CHAR(128) parameter to specify the owner name.

Result set

Column name	Data type	Description
TableName	CHAR(128)	The name of a table.
TableId	UNSIGNED INTEGER	The table ID.
IndexName	CHAR(128)	The name of an index.
IndexId	UNSIGNED INTEGER	The index ID. This column contains one of the following: ♦ **0** for primary keys ♦ **SYSFKEY.*foreign_key_id*** for foreign keys ♦ **SYSIDX.*index_id*** for all other indexes

Column name	Data type	Description
IndexType	CHAR(4)	The index type. This column contains one of the following values: ♦ **PKEY** for primary keys ♦ **FKEY** for foreign keys ♦ **UI** for unique indexes ♦ **UC** for unique constraints ♦ **NUI** for non-unique indexes
Levels	INTEGER	The number of levels in the index.

Remarks

The number of levels in the index tree determines the number of I/O operations needed to access a row using the index. Indexes with a small number of levels are more efficient than indexes with a large number of levels.

The procedure returns a result set containing the table name, the table ID, the index name, the index ID, the index type, and the number of levels in the index.

If no arguments are supplied, levels are returned for all indexes in the database. If only *tbl_name* is supplied, levels for all indexes on that table are supplied. If *tbl_name* is NULL and an *owner_name* is given, only levels for indexes on tables owned by that user are returned.

Permissions

DBA authority required

Side effects

None

See also

- "CREATE INDEX statement" on page 405
- "Using indexes" [*SQL Anywhere Server - SQL Usage*]

Example

The following example uses the sa_index_levels system procedure to return the number of levels in the Products index.

```
CALL sa_index_levels( );
```

TableName	TableId	IndexName	...	Levels
Products	436	Products	...	1
...	...	...	...	...

sa_java_loaded_classes system procedure

Lists the classes currently loaded by the database Java virtual machine.

Syntax

sa_java_loaded_classes( )

Result set

Column name	Data type	Description
class_name	VARCHAR(512)	The name of a class currently loaded by the database Java virtual machine.

Remarks

Returns a result set containing all the names of the Java classes currently loaded by the database Java virtual machine.

When the virtual machine is first called, it loads a number of classes. If you call sa_java_loaded_classes without using any Java in the database features beforehand, it returns this set of classes.

The procedure can be useful to diagnose missing classes. It can also be used to identify which classes from a particular jar are used by a given application.

Permissions

DBA authority required

Side effects

None

See also

- "Installing Java classes into a database" [*SQL Anywhere Server - Programming*].

sa_load_cost_model system procedure

Replaces the current cost model with the cost model stored in the specified file.

Syntax

sa_load_cost_model (*file_name* **)**

Arguments

- ***file_name*** Use this CHAR(1024) parameter to specify the name of the cost model file to load.

Remarks

The optimizer uses cost models to determine optimal access plans for queries. The database server maintains a cost model for each database. The cost model for a database can be recalibrated at any time using the

CALIBRATE SERVER clause of the ALTER DATABASE statement. For example, you might decide to recalibrate the cost model if you move the database onto non-standard hardware.

The sa_load_cost_model system procedure allows you to load a cost model that has been saved to file (*file_name*). Loading a cost model replaces the current cost model for the database.

Note
The sa_unload_cost_model system procedure does not include CALIBRATE PARALLEL READ information in the file that sa_load_cost_model loads.

Using the sa_load_cost_model system procedure can eliminate repetitive, time-consuming recalibration activities when there is a large number of identical hardware installations.

Exclusive use of the database is required when loading the new cost model.

When loading a cost model, consider whether it was generated for a database that is located on similar hardware. Loading a cost model from a database that is stored on significantly different hardware may cause poor performance due to inefficient access plans.

Cost models are saved to file using the sa_unload_cost_model system procedure. See "sa_unload_cost_model system procedure" on page 933.

Permissions

Must have DBA authority.

Side effects

The database server performs a COMMIT after loading the new cost model.

See also

- "ALTER DATABASE statement" on page 301
- "sa_unload_cost_model system procedure" on page 933
- "Query Optimization and Execution" [*SQL Anywhere Server - SQL Usage*]

Example

The following example loads the cost model from a file called costmodel8:

```
CALL sa_load_cost_model( costmodel8 );
```

sa_locks system procedure

Displays all locks in the database.

Syntax

sa_locks(
[*connection*
[, *creator*
[, *table_name*

[, *max_locks*]]]]
)

Arguments

- ***connection*** Use this INTEGER parameter to specify a connection ID. The procedure returns lock information only about the specified connection. The default value is 0 (or NULL), in which case information is returned about all connections.
- ***creator*** Use this CHAR(128) parameter to specify a user ID. The procedure returns information only about the tables owned by the specified user. The default value for the creator parameter is NULL. When this parameter is set to NULL, sa_locks returns the following information:
  - if the *table_name* parameter is unspecified, locking information is returned for all tables in the database
  - if the *table_name* parameter is specified, locking information is returned for tables with the specified name that were created by the current user
- ***table_name*** Use this CHAR(128) parameter to specify a table name. The procedure returns information only about the specified tables. The default value is NULL, in which case information is returned about all tables.
- ***max_locks*** Use this INTEGER parameter to specify the maximum number of locks for which to return information. The default value is 1000. The value -1 means return all lock information.

Result set

Column name	Data type	Description
conn_name	VARCHAR(128)	The name of the current connection.
conn_id	INTEGER	The ID number of the connection
user_id	CHAR(128)	The user connected through connection ID.
table_type	CHAR(6)	The type of table (either BASE or GBLTMP).
creator	VARCHAR(128)	The owner of the table.
table_name	VARCHAR(128)	The table on which the lock is held.
index_id	INTEGER	The index ID or NULL.
lock_class	CHAR(8)	The lock class. One of Schema, Row, Table, or Position. See "Objects that can be locked" [*SQL Anywhere Server - SQL Usage*].
lock_duration	CHAR(11)	The duration of the lock. One of Transaction, Position, or Connection.
lock_type	CHAR(9)	The lock type (this is dependent on the lock class).

Column name	Data type	Description
row_identifier	UNSIGNED BIGINT	The identifier for the row. This is either an 8-byte row identifier or NULL.

Remarks

The sa_locks procedure returns a result set containing information about all the locks in the database.

The value in the lock_type column depends on the lock classification in the lock_class column. The following values can be returned:

Lock class	Lock types	Comments
Schema	Shared (shared schema lock) Exclusive (exclusive schema lock)	For schema locks, the row_identifier and index ID values are NULL. See "Schema locks" [*SQL Anywhere Server - SQL Usage*].
Row	Read (read lock) Intent (intent lock) Write (write lock) Surrogate (surrogate lock)	Row read locks can be short-term locks (scans at isolation level 1) or can be long-term locks at higher isolation levels. The lock_duration column indicates whether the read lock is of short duration because of cursor stability (Position) or long duration, held until COMMIT/ROLLBACK (Transaction). Row locks are always held on a specific row, whose 8-byte row identifier is reported as a 64-bit integer value in the row_identifier column. A surrogate lock is a special case of a row lock. Surrogate locks are held on surrogate entries, which are created when referential integrity checking is delayed. See "Locking during inserts" [*SQL Anywhere Server - SQL Usage*]. There is not a unique surrogate lock for every surrogate entry created in a table. Rather, a surrogate lock corresponds to the set of surrogate entries created for a given table by a given connection. The row_identifier value is unique for the table and connection associated with the surrogate lock. See "Row locks" [*SQL Anywhere Server - SQL Usage*].
Table	Shared (shared table lock) Intent (intent to update table lock) Exclusive (exclusive table lock)	See "Table locks" [*SQL Anywhere Server - SQL Usage*].

Lock class	Lock types	Comments
Position	Phantom (phantom lock) Insert (insert lock)	In most cases, a position lock is also held on a specific row, and that row's 64-bit row identifier appears in the row_identifier column in the result set. However, Position locks can be held on entire scans (index or sequential), in which case the row_identifier column is NULL. See "Position locks" [*SQL Anywhere Server - SQL Usage*].

A position lock can be associated with a sequential table scan, or an index scan. The index_id column indicates whether the position lock is with respect to a sequential scan. If the position lock is held because of a sequential scan, the index_id column is NULL. If the position lock is held as the result of a specific index scan, the index identifier of that index is listed in the index_id column. The index identifier corresponds to the primary key of the ISYSIDX system table, which can be viewed using the SYSIDX view. If the position lock is held for scans over all indexes, the index ID value is -1.

Permissions

DBA authority required

Side effects

None

See also

- "How locking works" [*SQL Anywhere Server - SQL Usage*]
- "SYSIDX system view" on page 768

Example

For an example of this system procedure, as well as tips to augment the amount of information you can return, see "Obtaining information about locks" [*SQL Anywhere Server - SQL Usage*].

sa_make_object system procedure

Ensures that a skeletal instance of an object exists before executing an ALTER statement.

Syntax

sa_make_object(
objtype,
objname
[**,** *owner*
[**,** *tabname*]]
)

objtype:
'procedure'
| **'function'**

| **'view'**
| **'trigger'**
| **'service'**
| **'event'**

Arguments

- ***objtype*** Use this CHAR(30) parameter to specify the type of object being created.
- ***objname*** Use this CHAR(128) parameter to specify the name of the object to be created.
- ***owner*** Use this optional CHAR(128) parameter to specify the owner of the object to be created. The default value is CURRENT USER.
- ***tabname*** This CHAR(128) parameter is required only if objtype is 'trigger', in which case you use it to specify the name of the table on which the trigger is to be created.

Remarks

This procedure is particularly useful in scripts or command files that are run repeatedly to create or modify a database schema. A common problem in such scripts is that the first time they are run, a CREATE statement must be executed, but subsequent times an ALTER statement must be executed. This procedure avoids the necessity of querying the system views to find out whether the object exists.

To use the procedure, follow it by an ALTER statement that contains the entire object definition.

Permissions

Resource authority is required to create or modify database objects

Side effects

Automatic commit

See also

- “ALTER EVENT statement” on page 308
- “ALTER FUNCTION statement” on page 310
- “ALTER PROCEDURE statement” on page 315
- “ALTER TRIGGER statement” on page 341
- “ALTER VIEW statement” on page 342
- “ALTER SERVICE statement” on page 323

Examples

The following statements ensure that a skeleton procedure definition is created, define the procedure, and grant permissions on it. A command file containing these instructions could be run repeatedly against a database without error.

```
CALL sa_make_object( 'procedure','myproc' );
ALTER PROCEDURE myproc( in p1 INT, in p2 CHAR(30) )
BEGIN
    // ...
END;
GRANT EXECUTE ON myproc TO public;
```

The following example uses the sa_make_object system procedure to add a skeleton web service.


```
CALL sa_make_object( 'service','my_web_service' )
```

sa_materialized_view_info system procedure

Returns information about the specified materialized view.

Syntax

sa_materialized_view_info(
[*view_name*
[**,** *owner_name*]]
)

Arguments

- ***view_name*** Use this optional CHAR(128) parameter to specify the name of the materialized view for which to return information.
- ***owner_name*** Use this optional CHAR(128) parameter to specify the owner of the materialized view.

Remarks

If neither *view_name* nor *owner_name* are provided, information about all materialized views in the database is returned.

If *owner_name* is not provided, then only one of materialized views matching the specified view name is described. The procedure requires DBA authority or execute permissions on DBO owned procedures.

The sa_materialized_view_info system procedure returns the following information:

Column name	Data type	Description
OwnerName	CHAR(128)	The creator of the view.
ViewName	CHAR(128)	The name of the view.
Status	CHAR(1)	Status information about the view: ♦ D - disabled by the user, ♦ N - never refreshed ♦ E - error during the last refresh attempt ♦ F - underlying data has not changed since the last refresh (fresh) ♦ S - underlying data has changed since the last refresh (stale)
ViewLastRefreshed	TIMESTAMP	The time when the view was last refreshed. This value is NULL if the view has no data (uninitialized).
DataLastModified	TIMESTAMP	For a stale view, the last time that underlying data was modified.

Column name	Data type	Description
AvailForOptimization	CHAR(1)	Information about the availability of the view for use by the optimizer: ♦ Y - the view can be used by the optimizer ♦ D - use by optimizer disabled ♦ N - contains no data because a refresh has not been done or has failed ♦ I - the view cannot be used, for some internal reason ♦ O - incompatible option value for current connection

This procedure can be useful for determining the list of materialized views that will never be considered by the optimizer because of a problem with the definition of the view. The AvailForOptimization value is I for these materialized views. To learn more about the restrictions for materialized view definition, see "Restrictions when managing materialized views" [*SQL Anywhere Server - SQL Usage*].

Permissions

None

Side effects

All metadata for the specified materialized views, and all dependencies, are loaded into the server cache.

sa_migrate system procedure

Migrates a set of remote tables to a SQL Anywhere database.

Syntax

sa_migrate(
*base_table_owner***,**
server_name
[**,** *table_name*]
[**,** *owner_name*]
[**,** *database_name*]
[**,** *migrate_data*]
[**,** *drop_proxy_tables*]
[**,** *migrate_fkeys*]
)

Arguments

- ***base_table_owner*** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. Use the GRANT CONNECT statement to create this user. A value is required for this parameter. See "GRANT statement" on page 548.
- ***server_name*** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. Use the CREATE SERVER statement to create this server. A value is required for this parameter. See "CREATE SERVER statement" on page 435.

- ***table_name*** If you are migrating a single table, use this VARCHAR(128) parameter to specify the table name. Otherwise, you should specify NULL (the default) for this parameter. Do not specify NULL for both the *table_name* and *owner_name* parameters.
- ***owner_name*** If you are migrating only tables that belong to one owner, use this VARCHAR(128) parameter to specify the owner's name. Otherwise, you should enter NULL (the default) for this parameter. Do not specify NULL for both the *table_name* and *owner_name* parameters.
- ***database_name*** Use this VARCHAR(128) parameter to specify the name of the remote database. You must specify the database name if you want to migrate tables from only one database on the remote server. Otherwise, enter NULL (the default) for this parameter.
- ***migrate_data*** Use this optional BIT parameter to specify whether the data in the remote tables is migrated. This parameter can be 0 (do not migrate data) or 1 (migrate data). By default, data is migrated. (1)
- ***drop_proxy_tables*** Use this optional BIT parameter to specify whether the proxy tables created for the migration process are dropped once the migration is complete. This parameter can be 0 (proxy tables are not dropped) or 1 (proxy tables are dropped). By default, the proxy tables are dropped (1).
- ***migrate_fkeys*** Use this optional BIT parameter to specify whether the foreign key mappings are migrated. This parameter can be 0 (do not migrate foreign key mappings) or 1 (migrate foreign key mappings). By default, the foreign key mappings are migrated (1).

Remarks

You can use this procedure to migrate tables to SQL Anywhere from a remote Oracle, DB2, SQL Server, Adaptive Server Enterprise, or SQL Anywhere database. This procedure allows you to migrate in one step a set of remote tables, including their foreign key mappings, from the specified server. The sa_migrate system procedure calls the following system procedures:

- sa_migrate_create_remote_table_list
- sa_migrate_create_tables
- sa_migrate_data
- sa_migrate_create_remote_fks_list
- sa_migrate_create_fks
- sa_migrate_drop_proxy_tables

You might want to use these system procedures instead of sa_migrate if you need more flexibility. For example, if you are migrating tables with foreign key relationships that are owned by different users, you cannot retain the foreign key relationships if you use sa_migrate.

Before you can migrate any tables, you must first create a remote server to connect to the remote database using the CREATE SERVER statement. You may also need to create an external login to the remote database using the CREATE EXTERNLOGIN statement. See "CREATE SERVER statement" on page 435 and "CREATE EXTERNLOGIN statement" on page 397.

You can migrate all the tables from the remote database to a SQL Anywhere database by specifying only the *base_table_owner* and *server_name* parameters. However, if you specify only these two parameters, all the tables that are migrated will belong to one owner in the target SQL Anywhere database. If tables have different owners on the remote database and you want them to have different owners on the SQL Anywhere database, then you must migrate the tables for each owner separately, specifying the *base_table_owner* and *owner_name* parameters each time you call the sa_migrate procedure.

Caution
Do not specify NULL for both the *table_name* and *owner_name* parameters. Supplying NULL for both the *table_name* and *owner_name* parameters migrates all the tables in the database, including system tables. As well, tables that have the same name, but different owners in the remote database all belong to one owner in the target database. It is recommended that you migrate tables associated with one owner at a time.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_remote_fks_list system procedure" on page 892
- "sa_migrate_create_fks system procedure" on page 890
- "sa_migrate_drop_proxy_tables system procedure" on page 896

Examples

The following statement migrates all the tables belonging to user p_chin from the remote database, including foreign key mappings; migrates the data in the remote tables; and drops the proxy tables when migration is complete. In this example, all the tables that are migrated belong to local_user in the target SQL Anywhere database.

```
CALL sa_migrate( 'local_user', 'server_a', NULL, 'p_chin', NULL, 1, 1, 1 )
```

The following statement migrates only the tables that belong to user remote_a from the remote database. In the target SQL Anywhere database, these tables belong to the user local_a. Proxy tables created during the migration are not dropped at completion.

```
CALL sa_migrate( 'local_a', 'server_a', NULL, 'remote_a', NULL, 1, 0, 1 )
```

sa_migrate_create_fks system procedure

Creates foreign keys for each table listed in the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_fks(*i_table_owner* **)**

Arguments

- ***i_table_owner*** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated foreign keys. If you want to migrate tables that belong to different user, you must execute this procedure for each user whose tables you want to migrate. The *i_table_owner* is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 548.

Remarks

This procedure creates foreign keys for each table that is listed in the dbo.migrate_remote_fks_list table. The user specified by the *i_table_owner* argument owns the foreign keys in the target database.

If the tables in the target SQL Anywhere database do not all have the same owner, you must execute this procedure for each user who owns tables for which you need to migrate foreign keys.

Note

This system procedure is used in conjunction with several other migration system procedures, which must be executed in sequence as listed below:

1. sa_migrate_create_remote_table_list
2. sa_migrate_create_tables
3. sa_migrate_data
4. sa_migrate_create_remote_fks_list
5. sa_migrate_create_fks
6. sa_migrate_drop_proxy_tables

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_remote_fks_list system procedure" on page 892

- "sa_migrate_drop_proxy_tables system procedure" on page 896

Example

The following statement creates foreign keys based on the dbo.migrate_remote_fks_list table. The foreign keys belong to the user local_a on the local SQL Anywhere database.

```
CALL sa_migrate_create_fks( 'local_a' );
```

sa_migrate_create_remote_fks_list system procedure

Populates the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_remote_fks_list(*server_name* **)**

Arguments

- ***server_name*** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. The remote server is created with the CREATE SERVER statement. A value is required for this parameter. See "CREATE SERVER statement" on page 435.

Remarks

This procedure populates the dbo.migrate_remote_fks_list table with a list of foreign keys that can be migrated from the remote database. You can delete rows from this table for foreign keys that you do not want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 890.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_fks system procedure" on page 890
- "sa_migrate_drop_proxy_tables system procedure" on page 896

Example

The following statement creates a list of foreign keys that are in the remote database.

```
CALL sa_migrate_create_remote_fks_list( 'server_a' );
```

sa_migrate_create_remote_table_list system procedure

Populates the dbo.migrate_remote_table_list table.

Syntax

sa_migrate_create_remote_table_list(
i_server_name
[, *i_table_name*
[, *i_owner_name*
[, *i_database_name* **]]]**
)

Arguments

- ***i_server_name*** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. The remote server is created with the CREATE SERVER statement. A value is required for this parameter. See "CREATE SERVER statement" on page 435.
- ***i_table_name*** Use this optional VARCHAR(128) parameter to specify the name(s) of the tables that you want to migrate, or NULL to migrate all the tables. The default is NULL. Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters.
- ***i_owner_name*** Use this optional VARCHAR(128) parameter to specify the user who owns the tables on the remote database that you want to migrate, or NULL to migrate all the tables. The default is NULL. Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters
- ***i_database_name*** Use this optional VARCHAR(128) parameter to specify the name of the remote database from which you want to migrate tables. This parameter is NULL by default. When migrating tables from Adaptive Server Enterprise and Microsoft SQL Server databases, you must specify the database name.

Remarks

This procedure populates the dbo.migrate_remote_table_list table with a list of tables that can be migrated from the remote database. You can delete rows from this table for remote tables that you do not want to migrate.

If you do not want all the migrated tables to have the same owner on the target SQL Anywhere database, you must execute this procedure for each user whose tables you want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

Caution
Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters. Supplying NULL for both the *i_table_name* and *i_owner_name* parameters migrates all the tables in the database, including system tables. As well, tables that have the same name, but different owners in the remote database all belong to one owner in the target database. It is recommended that you migrate tables associated with one owner at a time.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 890.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_remote_fks_list system procedure" on page 892
- "sa_migrate_create_fks system procedure" on page 890
- "sa_migrate_drop_proxy_tables system procedure" on page 896

Example

The following statement creates a list of tables that belong to the user remote_a on the remote database.

```
CALL sa_migrate_create_remote_table_list( 'server_a', NULL, 'remote_a',
NULL );
```

sa_migrate_create_tables system procedure

Creates a proxy table and base table for each table listed in the dbo.migrate_remote_table_list table.

Syntax

sa_migrate_create_tables(*i_table_owner* **)**

Arguments

- ***i_table_owner*** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 548.

Remarks

This procedure creates a base table and proxy table for each table listed in the dbo.migrate_remote_table_list table (created using the sa_migrate_create_remote_table_list procedure). These proxy tables and base tables are owned by the user specified by the *i_table_owner* argument. This procedure also creates the same primary key indexes and other indexes for the new table that the remote table has in the remote database.

If you do not want all the migrated tables to have the same owner on the target SQL Anywhere database, you must execute the sa_migrate_create_remote_table_list procedure and the sa_migrate_create_tables procedure for each user who will own migrated tables.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 890.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_remote_fks_list system procedure" on page 892
- "sa_migrate_create_fks system procedure" on page 890
- "sa_migrate_drop_proxy_tables system procedure" on page 896

Example

The following statement creates base tables and proxy tables on the target SQL Anywhere database. These tables belong to the user local_a.

```
CALL sa_migrate_create_tables( 'local_a' );
```

sa_migrate_data system procedure

Migrates data from the remote database tables to the target SQL Anywhere database.

Syntax

sa_migrate_data(*i_table_owner* **)**

Arguments

- ***i_table_owner*** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 548.

Remarks

This procedure migrates the data from the remote database to the SQL Anywhere database for tables belonging to the user specified by the *i_table_owner* argument.

When the tables on the target SQL Anywhere database do not all have the same owner, you must execute this procedure for each user whose tables have data that you want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 890.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_create_remote_fks_list system procedure" on page 892
- "sa_migrate_create_fks system procedure" on page 890
- "sa_migrate_drop_proxy_tables system procedure" on page 896

Example

The following statement migrates data to the target SQL Anywhere database for tables that belong to the user local_a.

```
CALL sa_migrate_data( 'local_a' );
```

sa_migrate_drop_proxy_tables system procedure

Drops the proxy tables that were created for migration purposes.

Syntax

sa_migrate_drop_proxy_tables(*i_table_owner* **)**

Arguments

- ***i_table_owner*** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the proxy tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 548.

Remarks

This procedure drops the proxy tables that were created for the migration. The user who owns these proxy tables is specified by the *i_table_owner* argument.

If the migrated tables are not all owned by the same user on the target SQL Anywhere database, you must call this procedure for each user to drop all the proxy tables.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 890.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 888
- "sa_migrate_create_remote_table_list system procedure" on page 893
- "sa_migrate_create_tables system procedure" on page 894
- "sa_migrate_data system procedure" on page 895
- "sa_migrate_create_remote_fks_list system procedure" on page 892
- "sa_migrate_create_fks system procedure" on page 890

Example

The following statement drops the proxy tables on the target SQL Anywhere database that belong to the user local_a.

```
CALL sa_migrate_drop_proxy_tables( 'local_a' );
```

sa_performance_diagnostics system procedure

Returns a summary of request timing information for all connections when the database server has request timing logging enabled.

Syntax

sa_performance_diagnostics( )

Result set

Column name	Data type	Description
Number	INT	The ID number of the connection.
Name	VAR-CHAR (255)	The name of the connection.
Userid	VAR-CHAR (255)	The user ID for the connection.
DBNumber	INT	The ID number of the database.
LoginTime	TIMES-TAMP	The date and time the connection was established.
TransactionStartTime	TIMES-TAMP	The time the database was first modified after a COMMIT or ROLLBACK, or an empty string if no modifications have been made to the database since the last COMMIT or ROLL-BACK.
LastReqTime	TIMES-TAMP	The time at which the last request for the specified connection started.
ReqType	VAR-CHAR (255)	The type of the last request.
ReqStatus	VAR-CHAR (255)	The status of the request. It can be one of the following values: ♦ **Idle** The connection is not currently processing a request. ♦ **Unscheduled** The connection has work to do and is waiting for a worker thread. ♦ **BlockedIO** The connection is blocked waiting for an I/O. ♦ **BlockedContention** The connection is blocked waiting for access to shared database server data structures. ♦ **BlockedLock** The connection is blocked waiting for a locked object. ♦ **Executing** The connection is executing a request.
ReqTimeUnscheduled	DOUBLE	The time spent unscheduled.
ReqTimeActive	DOUBLE	The time spent waiting to process requests.

Column name	Data type	Description
ReqTimeBlockIO	DOUBLE	The time spent waiting for I/O to complete.
ReqTimeBlockLock	DOUBLE	The time spent waiting for a lock.
ReqTimeBlockContention	DOUBLE	The time spent waiting for atomic access.
ReqCountUnscheduled	INT	The number of times waited for scheduling.
ReqCountActive	INT	The number of requests processed.
ReqCountBlockIO	INT	The number of times waited for I/O to complete.
ReqCountBlockLock	INT	The number of times waited for a lock.
ReqCountBlockContention	INT	The number of times waited for atomic access.
LastIdle	INT	The number of ticks between requests.
BlockedOn	INT	If the current connection isn't blocked, this is zero. If it is blocked, the connection number on which the connection is blocked due to a locking conflict.
UncommitOp	INT	The number of uncommitted operations.
CurrentProcedure	VAR-CHAR (255)	The procedure that a connection is currently executing. If the connection is executing nested procedure calls, the name is the name of the current procedure. If there is no procedure executing, an empty string is returned
EventName	VAR-CHAR (255)	The name of the associated event if the connection is running an event handler. Otherwise, the result is NULL.
CurrentLineNumber	INT	The current line number of the procedure or compound statement a connection is executing. The procedure can be identified using the CurrentProcedure property. If the line is part of a compound statement from the client, an empty string is returned.
LastStatement	LONG VAR-CHAR	The most recently prepared SQL statement for the current connection.
LastPlanText	LONG VAR-CHAR	The long text plan of the last query executed on the connection.

Column name	Data type	Description
AppInfo	LONG VAR-CHAR	Information about the client that made the connection. For HTTP connections, this includes information about the browser. For connections using older versions of jConnect or Open Client, the information may be incomplete. The API value can be DBLIB, ODBC, OLEDB, or ADO.NET.
LockCount	INT	The number of locks held by the connection.
SnapshotCount	INT	The number of snapshots associated with the connection.

Remarks

The sa_performance_diagnostics system procedure returns a result set consisting of a set of request timing properties and statistics if the server has been told to collect the information. Recording of request timing information must be turned on the database server prior to calling sa_performance_diagnostics. To do this, specify the -zt option when starting the database server or execute the following:

```
CALL sa_server_option( 'RequestTiming','ON' );
```

Permissions

DBA authority required

Side effects

None

See also

- "-zt server option" [*SQL Anywhere Server - Database Administration*]
- "sa_performance_statistics system procedure" on page 901
- "sa_server_option system procedure" on page 914

Examples

You can execute the following query to identify connections that have spent a long time waiting for database server requests to complete.

```
SELECT  Number, Name,
      CAST( DATEDIFF( second, LoginTime, CURRENT TIMESTAMP ) AS DOUBLE ) AS
T,
      ReqTimeActive / T AS PercentActive
FROM  dbo.sa_performance_diagnostics()
WHERE PercentActive > 10.0
ORDER BY PercentActive DESC
```

Find all requests that are currently executing, and have been executing for more than 60 seconds:

```
SELECT  Number, Name,
     CAST( DATEDIFF( second, LastReqTime, CURRENT TIMESTAMP ) AS DOUBLE ) AS
ReqTime
FROM  dbo.sa_performance_diagnostics()
```



```
WHERE ReqStatus <> 'IDLE' AND ReqTime > 60.0
ORDER BY ReqTime DESC
```

sa_performance_statistics system procedure

Returns a summary of memory diagnostic statistics for all connections when the database server has request timing logging enabled.

Syntax

sa_performance_statistics( )

Result set

Column name	Data type	Description
DBNumber	INT	The ID number of the database.
ConnNumber	INT	An INTEGER representing a connection ID. Returns NULL if the Type is Server.
PropNum	INT	The connection property number.
PropName	VARCHAR (255)	The connection property name.
Value	INT	The connection property value.

Remarks

The sa_performance_statistics system procedure returns a result set consisting of a set of memory diagnostic statistics if the server has been told to collect the information. Recording of memory diagnostic statistics must be turned on the database server prior to calling sa_performance_statistics. To do this, specify the -zt option when starting the database server or execute the following:

```
CALL sa_server_option( 'RequestTiming','ON' );
```

Permissions

DBA authority required

Side effects

None

See also

- "-zt server option" [*SQL Anywhere Server - Database Administration*]
- "sa_performance_diagnostics system procedure" on page 897
- "sa_server_option system procedure" on page 914

Example

The following example unloads all performance statistics to a text file named *dump_stats.txt*:


```
UNLOAD
   SELECT CURRENT TIMESTAMP, *
   FROM sa_performance_statistics()
   TO 'dump_stats.txt'
   APPEND ON
```

sa_procedure_profile system procedure

Reports information about the execution time for each line within procedures, functions, events, or triggers that have been executed in a database. This procedure provides the same information as the Profile tab in Sybase Central.

Syntax

sa_procedure_profile(
[*filename*
[**,** *save_to_file*]]
)

Arguments

- ***filename*** Use this optional LONG VARCHAR(128) parameter to specify the file to which the profiling information should be saved, or from which file it should be loaded. See the Remarks section below for more about saving and loading the profiling information.
- ***save_to_file*** Use this optional INT(1) parameter to specify whether to save the profiling information to a file, or load it from a previously stored file.

Result set

Column name	Data type	Description
object_type	CHAR(1)	The type of object. See the Remarks section below for a list of possible object types.
object_name	CHAR(128)	The name of the stored procedure, function, event, or trigger. If the object_type is **C** or **D**, then this is the name of the foreign key for which the system trigger was defined.
owner_name	CHAR(128)	The object's owner.
table_name	CHAR(128)	The table associated with a trigger (the value is NULL for other object types).
line_num	UNSIGNED INTEGER	The line number within the procedure.
executions	UNSIGNED INTEGER	The number of times the line has been executed.
millisecs	UNSIGNED INTEGER	The time to execute the line, in milliseconds.
percentage	DOUBLE	The percentage of the total execution time required for the specific line.

Column name	Data type	Description
foreign_owner	CHAR(128)	The database user who owns the foreign table for a system trigger.
foreign_table	CHAR(128)	The name of the foreign table for a system trigger.

Remarks

You can use this procedure to:

- **Return detailed procedure profiling information** To do this, you can simply call the procedure without specifying any arguments.
- **Save detailed procedure profiling information to file** To do this, you must include the *filename* argument and specify 1 for the *save_to_file* argument.
- **Load detailed procedure profiling information from a previously saved file** To do this, you must include the *filename* argument and specify 0 for the *save_to_file* argument (or leave it off, since the default is 0). When using the procedure in this way, the loaded file must have been created by the same database as the one from which you are running the procedure; otherwise, the results may be unusable.

Since the result set includes information about the execution times for individual lines within procedures, triggers, functions, and events, as well as what percentage of the total procedure execution time those lines use, you can use this profiling information to fine-tune slower procedures that may decrease performance.

Before you can profile your database, you must enable profiling. See "Enabling procedure profiling" [*SQL Anywhere Server - SQL Usage*].

The object_type field of the result set can be:

- **P** stored procedure
- **F** function
- **E** event
- **T** trigger
- **C** ON UPDATE system trigger
- **D** ON DELETE system trigger

If you want summary information instead of line by line details for each execution, use the sa_procedure_profile_summary procedure instead.

Permissions

DBA authority required

Side effects

None

See also

- "sa_server_option system procedure" on page 914
- "sa_procedure_profile_summary system procedure" on page 904

Example

The following statement returns the execution time for each line of every procedure, function, event, or trigger that has been executed in the database:

```
CALL sa_procedure_profile( );
```

The following statement returns the same detailed procedure profiling information as the example above, and saves it to a file called *detailedinfo.txt*:

```
CALL sa_procedure_profile( "detailedinfo.txt", 1 );
```

Either of the following statements can be used to load detailed procedure profiling information from a file called *detailedinfoOLD.txt*:

```
CALL sa_procedure_profile( "detailedinfoOLD.txt", 0 );
```

```
CALL sa_procedure_profile( "detailedinfoOLD.txt" );
```

sa_procedure_profile_summary system procedure

Reports summary information about the execution times for all procedures, functions, events, or triggers that have been executed in a database. This procedure provides the same information for these objects as the Profile tab in Sybase Central.

Syntax

sa_procedure_profile_summary(
 [*filename*
 [, *save_to_file*]]
)

Arguments

- ***filename*** Use this optional LONG VARCHAR(128) parameter to specify the file to which the profiling information is saved, or from which file it should be loaded. See the Remarks section below for more about saving and loading the profiling information.
- ***save_to_file*** Use this optional INT(1) parameter to specify whether to save the summary information to a file, or to load it from a previously saved file.

Result set

Column name	Data type	Description
object_type	CHAR(1)	The type of object. See the Remarks section below for a list of possible object types.
object_name	CHAR(128)	The name of the stored procedure, function, event, or trigger.

Column name	Data type	Description
owner_name	CHAR(128)	The object's owner.
table_name	CHAR(128)	The table associated with a trigger (the value is NULL for other object types).
executions	UNSIGNED INTEGER	The number of times each procedure has been executed.
millisecs	UNSIGNED INTEGER	The time to execute the procedure, in milliseconds.
foreign_owner	CHAR(128)	The database user who owns the foreign table for a system trigger.
foreign_table	CHAR(128)	The name of the foreign table for a system trigger.

Remarks

You can use this procedure to:

- **Return current summary information** To do this, you can simply call the procedure without specifying any arguments.
- **Save current summary information to file** To do this, you must include the *filename* argument and specify 1 for the *save_to_file* argument.
- **Load stored summary information from a file** To do this, you must include the *filename* argument and specify 0 for the *save_to_file* argument (or leave it off, since the default is 0). When using the procedure in this way, the loaded file must have been created by the same database as the one from which you are running the procedure; otherwise, the results may be unusable.

Since the procedure returns information about the usage frequency and efficiency of stored procedures, functions, events, and triggers, you can use this information to fine-tune slower procedures to improve database performance.

Before you can profile your database, you must enable profiling. See "Enabling procedure profiling" [*SQL Anywhere Server - SQL Usage*].

The object_type field of the result set can be:

- **P** stored procedure
- **F** function
- **E** event
- **T** trigger
- **S** system trigger
- **C** ON UPDATE system trigger
- **D** ON DELETE system trigger

If you want line by line details for each execution instead of summary information, use the sa_procedure_profile procedure instead.

Permissions

DBA authority required

Side effects

None

See also

- "sa_server_option system procedure" on page 914
- "sa_procedure_profile system procedure" on page 902

Example

The following statement returns the execution time for any procedure, function, event, or trigger that has been executed in the database:

```
CALL sa_procedure_profile_summary( );
```

The following statement returns the same summary information as the previous example, and saves it to a file called *summaryinfo.txt*:

```
CALL sa_procedure_profile_summary( "summaryinfo.txt", 1 );
```

Either of the following statements can be used to load stored summary information from a file called *summaryinfoOLD.txt*:

```
CALL sa_procedure_profile_summary( "summaryinfoOLD".txt, 0 );
```

```
CALL sa_procedure_profile_summary( "summaryinfoOLD.txt" );
```

sa_recompile_views system procedure

Locates view definitions stored in the catalog that do not have column definitions and causes the column definitions to be created.

Syntax

sa_recompile_views([*ignore_errors*] **)**

Arguments

- ***ignore_errors*** Use this optional INTEGER parameter to specify whether to return errors during the recompilation. If you specify 0, an error is returned for each view for which column definition failed. If you specify 1, or any value other than 0, no errors are returned. If no value is specified, 0 is used by default.

Remarks

This procedure is used to locate views in the catalog that do not have column definitions and execute an ALTER VIEW statement with the RECOMPILE clause to create the column definitions. The procedure does this for each view that does not have a column definition until there are none left that require compilation

or until any remaining column definitions cannot be created. If the procedure is unable to recompile any views, an error is reported. Errors can be suppressed by specifying a non-zero parameter to this procedure.

Caution
The sa_recompile_views system procedure should only be called from within a *reload.sql* script. This procedure is used by the Unload utility (dbunload) and should not be used explicitly.

The sa_recompile_views system procedure does not attempt to recompile materialized views or any view marked DISABLED.

Permissions

DBA authority required

Side effects

For each non-materialized view that does not have a VALID status, an `ALTER VIEW` *owner*`.`*viewname* `ENABLE` statement is executed, causing an automatic commit.

See also

- "View status" [*SQL Anywhere Server - SQL Usage*]
- "force_view_creation option [database]" [*SQL Anywhere Server - Database Administration*]
- "ALTER VIEW statement" on page 342

sa_refresh_materialized_views system procedure

Initializes all materialized views that are in an uninitialized state.

Syntax

sa_refresh_materialized_views([*ignore_errors*] **)**

Arguments

- ***ignore_errors*** Use this optional INTEGER parameter to specify whether to return errors during the recompilation. If you specify 0, an error is returned for each view for which column definition failed. If you specify 1, or any value other than 0, no errors are returned. If no value is specified, 0 is used by default.

Remarks

A materialized view may be in an uninitialized state because it has just been created, has just been re-enabled, or the last attempt to initialize or refresh it failed due to an error. The sa_refresh_materialized_views system procedure scans the database for all such materialized views and attempts to initialize them. If the procedure encounters an error initializing a materialized view, it continues on attempting to process the remaining uninitialized views.

You can also use the REFRESH MATERIALIZED VIEW statement to initialize a materialized view.

Permissions

DBA authority required

Side effects

none

See also

- "REFRESH MATERIALIZED VIEW statement" on page 621
- "Refreshing materialized views" [*SQL Anywhere Server - SQL Usage*]

sa_remove_tracing_data system procedure

Permanently deletes from the diagnostic tracing tables all records pertaining to the specified logging (tracing) session ID.

Syntax

sa_remove_tracing_data(*log_session_id* **)**

Arguments

- ***log_session_id*** Use this INTEGER parameter to specify the ID of the logging session for which to remove the data.

Remarks

If there are no records for the specified *log_session_id*, the procedure has no effect. The procedure has no return values.

Permissions

DBA authority required.

Side effects

Causes a commit upon completion, even if no records were found for the specified *log_session_id*.

See also

- "Diagnostic tracing tables" on page 735

sa_report_deadlocks system procedure

Retrieves information about deadlocks from an internal buffer created by the database server.

Syntax

sa_report_deadlocks( )

Result set

Column name	Data type	Description
snapshotId	bigint	The deadlock instance (all rows pertaining to a particular deadlock have the same ID).

Column name	Data type	Description
snapshotAt	TIMESTAMP	The time when the deadlock occurred.
waiter	INT	The connection handle of the waiting connection.
who	VARCHAR(128)	The user ID associated with the connection that is waiting.
what	LONG VARCHAR	The command being executed by the waiting connection. This information is only available if you have turned on capturing of the most recently-prepared SQL statement by specifying the -zl option on the database server command line or have turned this feature on using the sa_server_option system procedure.
wait_on	bigint	The name of the lock being waited on.
owner	INT	The connection handle of the connection owning the lock being waited on.

Remarks

When the log_deadlocks option is set to On, the database server logs information about deadlocks in an internal buffer. You can view the information in the log using the sa_report_deadlocks system procedure.

Permissions

DBA authority required

Side effects

None

See also

- "log_deadlocks option [database]" [*SQL Anywhere Server - Database Administration*]
- "Determining who is blocked" [*SQL Anywhere Server - SQL Usage*]

sa_reset_identity system procedure

Allows the next identity value to be set for a table. Use this to change the autoincrement value for the next row that will be inserted.

Syntax

sa_reset_identity(
tbl_name,
owner_name,
new_identity
)

Arguments

- ***tbl_name*** Use this CHAR(128) parameter to specify the table for which you want to reset the identity value. If the owner of the table is not specified, *tbl_name* must uniquely identify a table in the database.
- ***owner_name*** Use this CHAR(128) parameter to specify the owner of the table for which you want to reset the identity value.
- ***new_identity*** Use this BIGINT parameter to specify the value from which you want the auto-incrementing to start.

Remarks

The next identity value generated for a row inserted into the table is *new_identity* + 1.

No checking occurs to see whether *new_identity* + 1 conflicts with existing rows in the table. For example, if you specify *new_identity* as 100, the next row inserted gets an identity value of 101. However, if 101 already exists in the table, the row insertion fails.

If *owner* is not specified or is NULL, *tbl_name* must uniquely identify a table in the database.

Permissions

DBA authority required

Side effects

Causes a checkpoint to occur after the value has been updated

Example

The following statement resets the next identity value to 101:

```
CALL sa_reset_identity( 'Employees', 'DBA', 100 );
```

sa_rowgenerator system procedure

Returns a result set with rows between a specified start and end value.

Syntax

sa_rowgenerator(
[*rstart*
[**,** *rend*
[**,** *rstep*]]]
)

Arguments

- ***rstart*** Use this optional INTEGER parameter to specify the starting value. The default value is 0.
- ***rend*** Use this optional INTEGER parameter to specify the ending value. The default value is 100.
- ***rstep*** Use this optional INTEGER parameter to specify the increment by which the sequence values are increased. The default value is 1.

Result set

Column name	Data type	Description
row_num	INTEGER	Sequence number.

Remarks

The sa_rowgenerator procedure can be used in the FROM clause of a query to generate a sequence of numbers. This procedure is an alternative to using the RowGenerator system table. You can use sa_rowgenerator for such tasks as:

- generating test data for a known number of rows in a result set.
- generating a result set with rows for values in every range. For example, you can generate a row for every day of the month, or you can generate ranges of zip codes.
- generating a query that has a specified number of rows in the result set. This may be useful for testing the performance of queries.

You can emulate the behavior of the RowGenerator table with the following statement:

```
SELECT row_num FROM sa_rowgenerator( 1, 255 );
```

Permissions

None

Side effects

None

See also

- "RowGenerator table (dbo)" on page 751

Example

The following query returns a result set containing one row for each day of the current month.

```
SELECT DATEADD( day, row_num-1,
        YMD( DATEPART( year, CURRENT DATE ),
             DATEPART( month, CURRENT DATE ), 1 ) )
    AS day_of_month
    FROM sa_rowgenerator( 1, 31, 1 )
    WHERE DATEPART( month, day_of_month ) =
       DATEPART( month, CURRENT DATE )
    ORDER BY row_num;
```

The following query shows how many employees live in zip code ranges (0-9999), (10000-19999), ..., (90000-99999). Some of these ranges have no employees, which causes the warning `Null value eliminated in aggregate function (-109)`. The sa_rowgenerator procedure can be used to generate these ranges, even though no employees have a zip code in the range.

```
SELECT row_num AS r1, row_num+9999
 AS r2, COUNT( PostalCode ) AS zips_in_range
FROM sa_rowgenerator( 0, 99999, 10000 ) D LEFT JOIN Employees
   ON PostalCode BETWEEN r1 AND r2
```



```
GROUP BY r1, r2
ORDER BY 1;
```

The following example generates 10 rows of data and inserts them into the NewEmployees table:

```
INSERT INTO NewEmployees ( ID, Salary, Name )
SELECT row_num,
   CAST( RAND() * 1000 AS INTEGER ),
   'Mary'
FROM sa_rowgenerator( 1, 10 );
```

The following example uses the sa_rowgenerator system procedure to create a view containing all integers. The value 2147483647 in this example represents the maximum signed integer supported in SQL Anywhere.

```
CREATE VIEW Integers AS
SELECT row_num AS n
FROM sa_rowgenerator( 0, 2147483647, 1 );
```

This example uses the sa_rowgenerator system procedure to create a view containing dates from 0001-01-01 to 9999-12-31. The value 3652058 in this example represents the number of days between 0001-01-01 and 9999-12-31, the earliest and latest dates supported in SQL Anywhere.

```
CREATE VIEW Dates AS
SELECT DATEADD( day, row_num, '0001-01-01' ) AS d
FROM sa_rowgenerator( 0, 3652058, 1 );
```

sa_save_trace_data system procedure

Saves tracing data to base tables.

Syntax

sa_save_trace_data( )

Remarks

While a tracing session is running, diagnostic data is stored in temporary versions of the diagnostic tracing tables. When you stop a tracing session, you specify whether you want to permanently store the tracing data in the base tables for diagnostic tracing. If you do not choose to save the data, you can still save the data after the session is stopped by using the sa_save_trace_data system procedure.

The sa_save_trace_data system procedure returns an error if tracing is still in progress; you must stop tracing in order to use this system procedure.

The sa_save_trace_data system procedure can be used even if the user specified WITHOUT SAVING when stopping tracing. Also, the procedure must be called from the tracing database.

Permissions

DBA authority required

Side effects

Automatic commit.

See also

- "Creating a tracing session" [*SQL Anywhere Server - SQL Usage*]
- "Diagnostic tracing tables" on page 735

sa_send_udp system procedure

Sends a UDP packet to the specified address.

Syntax

sa_send_udp(
destAddress,
destPort,
msg
)

Arguments

- ***destAddress*** Use this CHAR(254) to specify either the host name or IP number.
- ***destPort*** Use this UNSIGNED SMALLINT parameter to specify the port number to use.
- ***msg*** Use this LONG BINARY parameter to specify the message to send to the specified address. If this value is a string, it must be enclosed in single quotes.

Remarks

This procedure sends a single UDP packet to the specified address. The procedure returns 0 if the message is sent successfully, and returns an error code if an error occurs. The error code is one of the following:

- -1 if the message is too large to send over a UDP socket (as determined by the operating system) or if there is a problem with the destination address
- the Winsock/Posix error code that is returned by the operating system

If the *msg* parameter contains binary data or is more complex than a string, you may want to use a variable. For example,

```
CREATE VARIABLE v LONG BINARY;
SET v='This is a UDP message';
SELECT dbo.sa_send_udp( '10.25.99.124', 1234, v );
DROP VARIABLE v;
```

This procedure can be used with MobiLink server-initiated synchronization to wake up the Listener utility (*dblsn.exe*). If you use the sa_send_udp system procedure as a way to notify the Listener, you should append a 1 to your UDP packet. This number is a server-initiated synchronization protocol number. In future versions of MobiLink, new protocol versions may cause the Listener to behave differently.

Permissions

DBA authority required

Side effects

None

See also

- "Notifying the Listener with sa_send_udp" [*MobiLink - Server-Initiated Synchronization*]

Example

The following example sends the message "This is a test" to IP address 10.25.99.196 on port 2345:

```
CALL sa_send_udp( 10.25.99.196, 2345, 'This is a test' );
```

sa_server_option system procedure

Overrides a server option while the server is running.

Syntax

sa_server_option(
opt,
val
)

Arguments

- ***opt*** Use this CHAR(128) parameter to specify a server option name.
- ***val*** Use this CHAR(128) parameter to specify the new value for the server option.

Remarks

Database administrators can use this procedure to override some database server options temporarily without restarting the database server.

The option values that are changed using this procedure are reset to their default values when the server shuts down. If you want to change an option value every time the server is started, you can specify the corresponding server option when the database server is started if one exists.

The following option settings can be changed:

Option name	Values	Default	Server option
CacheSizingStatistics	YES, NO	NO	"-cs server option" [*SQL Anywhere Server - Database Administration*]
CollectStatistics	YES, NO	YES	"-k server option" [*SQL Anywhere Server - Database Administration*]
ConnsDisabled	YES, NO	NO	
ConnsDisabledForDB	YES, NO	NO	

Option name	Values	Default	Server option
ConsoleLogFile	*filename*		"-o server option" [*SQL Anywhere Server - Database Administration*]
ConsoleLogMaxSize	*file-size*, in bytes		"-on server option" [*SQL Anywhere Server - Database Administration*]
DatabaseCleaner	ON, OFF	ON	
DebuggingInformation	YES, NO	NO	"-z server option" [*SQL Anywhere Server - Database Administration*]
IdleTimeout	INTEGER, in minutes	240	"-ti server option" [*SQL Anywhere Server - Database Administration*]
LivenessTimeout	INTEGER, in seconds	120	"-tl server option" [*SQL Anywhere Server - Database Administration*]
ProcedureProfiling	YES, NO, RESET, CLEAR	NO	
ProfileFilterConn	*connection-id*		
ProfileFilterUser	*user-id*		
QuittingTime	valid date and time		"-tq server option" [*SQL Anywhere Server - Database Administration*]
RememberLastPlan	YES, NO	NO	"-zp server option" [*SQL Anywhere Server - Database Administration*]
RememberLastStatement	YES, NO	NO	"-zl server option" [*SQL Anywhere Server - Database Administration*]
RequestFilterConn	*connection-id*, -1		
RequestFilterDB	*database-id*, -1		
RequestLogFile	*filename*		"-zo server option" [*SQL Anywhere Server - Database Administration*]

Option name	Values	Default	Server option
RequestLogging	SQL, HOSTVARS, PLAN, PROCEDURES, TRIGGERS, OTHER, BLOCKS, REPLACE, ALL, YES, NONE, NO	NONE	"-zr server option" [*SQL Anywhere Server - Database Administration*]
RequestLogMaxSize	*file-size*, in bytes		"-zs server option" [*SQL Anywhere Server - Database Administration*]
RequestLogNumFiles	INTEGER		"-zn server option" [*SQL Anywhere Server - Database Administration*]
RequestTiming	YES, NO	NO	"-zt server option" [*SQL Anywhere Server - Database Administration*]
SecureFeatures	*feature-list*		"-sf server option" [*SQL Anywhere Server - Database Administration*]

CacheSizingStatistics
When set to YES, display cache information in the Server Messages window whenever the cache size changes. See "-cs server option" [*SQL Anywhere Server - Database Administration*].

CollectStatistics
When set to YES, the database server collects Performance Monitor statistics. See "-k server option" [*SQL Anywhere Server - Database Administration*].

ConnsDisabled
When set to YES, no other connections are allowed to any databases on the database server.

ConnsDisabledForDB
When set to YES, no other connections are allowed to the current database.

ConsoleLogFile
The name of the file used to record Server Messages window information. Specifying an empty string stops logging to the file. Any backslash characters in the path must be doubled because this is a SQL string. See "-o server option" [*SQL Anywhere Server - Database Administration*].

ConsoleLogMaxSize
The maximum size, in bytes, of the file used to record Server Messages window information. When the output log file reaches the size specified by either the sa_server_option system procedure or the -on server option, the file is renamed with the extension *.old* appended (replacing an existing file with the same name if one exists). The output log file is then restarted. See "-on server option" [*SQL Anywhere Server - Database Administration*].

DatabaseCleaner

Do not change the setting of this option except on the recommendation of iAnywhere Technical Support. See also "sa_clean_database system procedure" on page 842.

DebuggingInformation
Displays diagnostic messages and other messages for troubleshooting purposes. The messages appear in the Server Messages window. See "-z server option" [*SQL Anywhere Server - Database Administration*].

IdleTimeout
Disconnects TCP/IP or SPX connections that have not submitted a request for the specified number of minutes. This prevents inactive connections from holding locks indefinitely. See "-ti server option" [*SQL Anywhere Server - Database Administration*].

LivenessTimeout
A liveness packet is sent periodically across a client/server TCP/IP or SPX network to confirm that a connection is intact. If the network server runs for a LivenessTimeout period without detecting a liveness packet, the communication is severed. See "-tl server option" [*SQL Anywhere Server - Database Administration*].

ProcedureProfiling
Controls procedure profiling for stored procedures, functions, events, and triggers. Procedure profiling shows you how long it takes your stored procedures, functions, events, and triggers to execute. You can also set procedure profiling options on the Database property sheet in Sybase Central.

- **YES** enables procedure profiling for the database you are currently connected to.
- **NO** disables procedure profiling and leaves the profiling data available for viewing.
- **RESET** returns the profiling counters to zero, without changing the YES or NO setting.
- **CLEAR** returns the profiling counters to zero and disables procedure profiling.

Once profiling is enabled, you can use the sa_procedure_profile_summary and sa_procedure_profile system procedures to retrieve profiling information from the database. See "Procedure profiling using system procedures" [*SQL Anywhere Server - SQL Usage*].

ProfileFilterConn
Instructs the database server to capture profiling information for a specific connection ID, without preventing other connections from using the database. When connection filtering is enabled, the value returned for `SELECT property( 'ProfileFilterConn' )` is the connection ID of the connection being monitored. If no ID has been specified, or if connection filtering is disabled, the value returned is -1.

ProfileFilterUser
Instructs the database server to capture profiling information for a specific user ID.

QuittingTime
Instructs the database server to shut down at the specified time. See "-tq server option" [*SQL Anywhere Server - Database Administration*].

RememberLastPlan property

Instructs the database server to capture the long text plan of the last query executed on the connection. This setting is also controlled by the -zp server option. See "-zp server option" [*SQL Anywhere Server - Database Administration*].

You can obtain the current value of the LastPlan for a connection by querying the value of the LastPlanText connection property:

```
SELECT CONNECTION_PROPERTY( 'LastPlanText' )
```

RememberLastStatement
Instructs the database server to capture the most recently prepared SQL statement for each database running on the server. For stored procedure calls, only the outermost procedure call appears, not the statements within the procedure.

You can obtain the current value of the LastStatement for a connection by querying the value of the LastStatement connection property:

```
SELECT CONNECTION_PROPERTY( 'LastStatement' )
```

For more information, see "Server-level properties" [*SQL Anywhere Server - Database Administration*] and "-zl server option" [*SQL Anywhere Server - Database Administration*].

When RememberLastStatement is turned on, the following statement returns the most recently-prepared statement for the specified connection.

```
SELECT CONNECTION_PROPERTY( 'LastStatement', connection-id )
```

The sa_conn_activity system procedure returns this same information for all connections.

Caution
When -zl is specified, or when the RememberLastStatement server setting is turned on, any user can call the sa_conn_activity system procedure or obtain the value of the LastStatement connection property to find out the most recently-prepared SQL statement for any other user. This option should be used with caution and turned off when it is not required.

RequestFilterConn
Filter the request logging information so that only information for a particular connection is logged. This can help reduce the size of the request log file when monitoring a database server with many active connections or multiple databases. You can obtain the connection ID by executing the following:

```
CALL sa_conn_info( )
```

To specify a specific connection to be logged once you have obtained the connection ID, execute the following:

```
CALL sa_server_option( 'RequestFilterConn', connection-id )
```

Filtering remains in effect until it is explicitly reset, or until the database server is shut down. To reset filtering, use the following statement:

```
CALL sa_server_option( 'RequestFilterConn', -1 )
```

RequestFilterDB

Filter the request logging information so that only information for a particular database is logged. This can help reduce the size of the request log file when monitoring a server with multiple databases. You can obtain the database ID by executing the following statement when you are connected to the desired database:

```
SELECT connection_property( 'DBNumber' )
```

To specify that only information for a particular database is to be logged, execute the following:

```
CALL sa_server_option( 'RequestFilterDB', database-id )
```

Filtering remains in effect until it is explicitly reset, or until the database server is shut down. To reset filtering, use the following statement:

```
CALL sa_server_option( 'RequestFilterDB', -1 )
```

RequestLogFile
The name of the file used to record request information. Specifying an empty string stops logging to the request log file. If request logging is enabled but the request log file was not specified or has been set to an empty string, the server logs requests to the Server Messages window. Any backslash characters in the path must be doubled as this is a SQL string. See "-zo server option" [*SQL Anywhere Server - Database Administration*].

RequestLogging
This call turns on logging of individual SQL statements sent to the database server for use in troubleshooting, in conjunction with the database server -zr and -zo options. Values can be combinations of the following, separated by either a plus sign (+), or a comma:

- **SQL** enables logging of the following:
  - START DATABASE statements
  - STOP DATABASE statements
  - STOP ENGINE statements
  - Statement preparation and execution
  - EXECUTE IMMEDIATE statement
  - Option settings
  - COMMIT statements
  - ROLLBACK statements
  - PREPARE TO COMMIT operations
  - Connects and disconnects
  - Beginnings of transactions
  - DROP STATEMENT statements
  - Cursor explanations
  - Cursor open, close, and resume
  - Errors
- **PLAN** enables logging of query plans (short form). Query plans for procedures are also recorded if logging of procedures (PROCEDURES) is enabled.
- **HOSTVARS** enables logging of host variable values. If you specify HOSTVARS, the information listed for SQL is also logged.

- **PROCEDURES** enables logging of statements executed from within procedures.
- **TRIGGERS** enables logging of statements executed from within triggers.
- **OTHER** enables logging of additional request types not included by SQL, such as FETCH and PREFETCH. However, if you specify OTHER but do not specify SQL, it is the equivalent of specifying SQL+OTHER. Including OTHER can cause the log file to grow rapidly and could negatively impact server performance.
- **BLOCKS** enables logging of details showing when a connection is blocked and unblocked on another connection.
- **REPLACE** at the start of logging, the existing request log is replaced with a new (empty) one of the same name. Otherwise, the existing request log is opened and new entries are appended to the end of the file.
- **ALL** logs all supported information. This is equivalent to specifying SQL+PLAN+HOSTVARS +PROCEDURES+TRIGGERS+OTHER+BLOCKS. This setting can cause the log file to grow rapidly and could negatively impact server performance.
- **NO or NONE** turns off logging to the request log.

You can find the current value of the RequestLogging setting using `SELECT property ( 'RequestLogging' )`.

For more information, see "-zr server option" [*SQL Anywhere Server - Database Administration*], and "Server-level properties" [*SQL Anywhere Server - Database Administration*].

RequestLogMaxSize
The maximum size of the file used to record request logging information, in bytes. If you 0, then there is no maximum size for the request logging file, and the file is never renamed. This is the default value.

When the request log file reaches the size specified by either the sa_server_option system procedure or the -zs server option, the file is renamed with the extension *.old* appended (replacing an existing file with the same name if one exists). The request log file is then restarted. See "-zs server option" [*SQL Anywhere Server - Database Administration*].

RequestLogNumFiles
The number of request log file copies to retain.

If request logging is enabled over a long period of time, the request log file can become large. The –zn option allows you to specify the number of request log file copies to retain. See "-zn server option" [*SQL Anywhere Server - Database Administration*].

RequestTiming
Instructs the database server to maintain timing information for each connection. This feature is turned off by default. When it is turned on, the database server maintains cumulative timers for each connection that indicate how much time the connection spent in the server in each of several states. You can use the sa_performance_diagnostics system procedure to obtain a summary of this timing information, or you can retrieve individual values by inspecting the following connection properties:

- ReqCountUnscheduled

- ReqTimeUnscheduled
- ReqCountActive
- ReqTimeActive
- ReqCountBlockIO
- ReqTimeBlockIO
- ReqCountBlockLock
- ReqTimeBlockLock
- ReqCountBlockContention
- ReqTimeBlockContention

See "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

When the RequestTiming server property is on, there is a small overhead for each request to maintain the additional counters. See "-zt server option" [*SQL Anywhere Server - Database Administration*], and "sa_performance_diagnostics system procedure" on page 897.

SecureFeatures
Specifies features that are disabled for databases running on this database server. The *feature-list* is a comma-separated list of feature names or feature sets. For a list of valid *feature-list* values, see "-sf server option" [*SQL Anywhere Server - Database Administration*].

Any changes you make to enable or disable features take effect immediately. The settings do not affect the connection that executes the sa_server_option system procedure.

To use the sa_server_option system procedure to enable or disable features for all databases running on the current database server, you must specify a key with the -sk option when starting the database server, and then set the value of the secure_feature_key database option to the key specified by -sk. Setting the secure_feature_key database option to the -sk value enables all features for the current connection. See "-sk server option" [*SQL Anywhere Server - Database Administration*] and "secure_feature_key [database]" [*SQL Anywhere Server - Database Administration*].

Permissions

DBA authority required

Side effects

None

Example

The following statement disallows new connections to the database server.

```
CALL sa_server_option( 'ConnsDisabled', 'YES' );
```

The following statement disallows new connections to the current database.

```
CALL sa_server_option( 'ConnsDisabledForDB', 'YES' );
```

The following statement enables logging of all SQL statements, procedure calls, plans, blocking and unblocking events, and specifies that a new request log be started.

```
CALL dbo.sa_server_option( 'RequestLogging', 'SQL+PROCEDURES+BLOCKS+PLAN
+REPLACE' );
```


sa_set_http_header system procedure

Permits a web service to set an HTTP header in the result.

Syntax

sa_set_http_header(
fldname,
val
)

Arguments

- ***fldname*** Use this CHAR(128) parameter to specify a string containing the name of one of the HTTP header fields.
- ***val*** Use this LONG VARCHAR parameter to specify the value to which the named parameter should be set.

Remarks

Setting the special header field @HTTPSTATUS sets the status code returned with the request. The status code is also known as the response code. For example, the following command sets the status code to 404 Not Found.

```
CALL dbo.sa_set_http_header( '@HTTPSTATUS', '404' );
```

The body of the error message is inserted automatically. Only valid HTTP error codes can be used. Setting the status to an invalid code causes a SQL error.

Permissions

None

Side effects

None

See also

- "sa_split_list system procedure" on page 927

Example

The following example sets the Content-Type header field to text/html.

```
CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
```

sa_set_http_option system procedure

Permits a web service to set an HTTP option in the result.

Syntax

sa_set_http_option(
optname,

val
)

Arguments

- ***optname*** Use this CHAR(128) parameter to specify a string containing the name of one of the HTTP options.
- ***val*** Use this LONG VARCHAR parameter to specify the value to which the named option should be set.

Remarks

Use this procedure within statements or procedures that handle web services to set options within an HTTP result set.

Following are the supported options:

- **CharsetConversion** Use this option to control whether the result set is to be automatically converted from the character set of the database to the character set of the client. The only permitted values are ON and OFF. The default value is ON. See "Using automatic character set conversion" [*SQL Anywhere Server - Programming*].
- **AcceptCharset** Use this option to specify the HTTP server preferences for the character sets used in an XML, SOAP, and HTTP web service. The syntax for this option conforms to the syntax used for the HTTP Accept-Charset request-header field specification in RFC2616 Hypertext Transfer Protocol. That is, permitted values are character set labels. For example, `sa_set_http_option ( 'AcceptCharset', 'UTF-8, Shift_JIS' )`. You may also include a quality value (q) for a character set label, indicating the degree of preference. For example, `sa_set_http_option ( 'AcceptCharset', 'UTF-8; q=0.9, Shift_JIS' )`.

  The final list of acceptable character sets is an intersection of the values in the client's Accept-Charset HTTP request-header field, and the values in the server AcceptCharset option. The quality values specified in the client request are respected such that the most preferred character set specified by the client is used when possible. Following are some possible scenarios for determining the list of acceptable character sets:

  - If the client request does not specify a Accept-Charset value, the server AcceptCharset option is used to determine the character set.
  - If the server AcceptCharset option is not specified, the client Accept-Charset request-header field value is used to determine the character set.
  - If neither the server AcceptCharset option, nor the client Accept-Charset request-header field, are specified, the database character set is used. For a SOAP request, the encoding used in the request is also used for the response.
  - If both the server AcceptCharset option, and the client Accept-Charset request-header field, are specified, then:

    1. If a match is found, that character set is used. If more than one match is found, the character set with the highest q-value in the client Accept-Charset request-header field is used.

2. If no match is found, the character set specified in client's Accept-Charset request-header field is used.

SQL Anywhere also allows you to specify a plus sign (+) to indicate the server preference for using the database character set if possible. For example, `sa_set_http_option( 'AcceptCharset', 'UTF-8, +' )`. When specified, if the list of character sets in the client Accept-Charset request-header field includes the database character set, that character set is automatically used, regardless of any quality values, and regardless of the order of matches found. This minimizes the need for character set conversion.

- **sessionid** Use this option to supply a name for an HTTP session. For example, `sa_set_http_option( 'sessionid', 'my_app_session_1' )` sets the ID for an HTTP session to my_app_session_1. The session ID must be a non-NULL string. For more information about HTTP sessions, see "Using HTTP sessions" [*SQL Anywhere Server - Programming*].

Permissions

None

Side effects

None

See also

- "SQL Anywhere Web Services" [*SQL Anywhere Server - Programming*]
- "Using HTTP sessions" [*SQL Anywhere Server - Programming*]
- "sa_set_http_header system procedure" on page 922

sa_set_soap_header system procedure

Permits the setting of SOAP headers for SOAP responses. This procedure is used within stored procedures called from SOAP web services.

Syntax

sa_set_soap_header(
fldname,
val
)

Arguments

- ***fldname*** Use this VARCHAR parameter to specify the header key, a unique string used to reference the given header entry (it need not be identical to the localname of the *val*).
- ***val*** Use this VARCHAR parameter to specify the raw XML of a top level header entry and its children within the scope of a SOAP Header element.

Remarks

All SOAP header entries set with this procedure are serialized within the SOAP Header element when the SOAP response message is sent. A *val* of NULL is not serialized. If no header entries exist for a SOAP response, then an enclosing Header element, within the SOAP envelope, is not created.

Permissions

None

Side effects

None

See also

- "SOAP_HEADER function [SOAP]" on page 248
- "NEXT_SOAP_HEADER function [SOAP]" on page 209
- "Working with SOAP headers" [*SQL Anywhere Server - Programming*]

Example

The following example sets the SOAP header welcome to Hello:

```
sa_set_soap_header( 'welcome', '<welcome>Hello</welcome>' )
```

sa_set_tracing_level system procedure

Initializes the level of tracing information to be stored in the diagnostic tracing tables.

Syntax

sa_set_tracing_level(
level
[**,** *specified_scope*
, *specified_name*]
[**,** *do_commit*]
)

Arguments

- ***level*** Use this INTEGER parameter to specify the level of diagnostic tracing to perform. Possible values include:
  - **0** Do not generate any tracing data. This level keeps the tracing session open, but does not send any tracing data to the diagnostic tracing tables.
  - **1** Sets a basic level of tracing.
  - **2** Sets a medium level of tracing.
  - **3** Sets a high level of tracing.
- ***specified_scope*** Use this optional LONG VARCHAR parameter to specify the tracing scope; for example, USER, DATABASE, CONNECTION_NAME, TRIGGER, and so on.

- ***specified_name*** Use this optional LONG VARCHAR parameter to specify the identifier for the object indicated in *specified_scope*.
- ***do_commit*** Use this optional TINYINT parameter to specify whether to commit, automatically, rows inserted by this procedure. Specify 1 (the default) to commit the rows automatically (recommended), and 0 to not commit them automatically.

Remarks

This procedure replaces the rows into the sa_diagnostic_tracing_level table, changing the tracing level and scope to the settings specified when calling the procedure.

Setting the level 0 does not stop the tracing session. Instead, the tracing session remains attached to the tracing database, but no tracing data is sent. The tracing session is still active when the level is 0.

This system procedure must be called from the database being profiled.

Permissions

DBA authority required

Side effects

None.

See also

- "Choosing a tracing level" [*SQL Anywhere Server - SQL Usage*]
- "Diagnostic tracing scopes" [*SQL Anywhere Server - SQL Usage*]
- "sa_diagnostic_tracing_level table" on page 748
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

Examples

The following example sets the tracing level to 1. This means that the entire database will be profiled for performance counter data, as well as some samples of executed statements:

```
CALL sa_set_tracing_level( 1 );
```

The following example sets the tracing level to 3, and specifies the user AG84756. This means that only activities associated with AG84756 will be traced:

```
CALL sa_set_tracing_level( 3, 'user', 'AG84756' );
```

sa_snapshots system procedure

Returns a list of snapshots that are currently active.

Syntax

sa_snapshots( )

Result set

Column name	Data type	Description
connection_num	INT	The connection ID for the connection on which the snapshot is running.
start_sequence_num	UNSIGNED BIG-INT	A unique number that identifies the snapshot.
statement_level	BIT	True if the snapshot was created with state-ment-snapshot or readonly-statement-snap-shot. Otherwise, false.

Remarks

Several statement snapshots can exist on one connection. In the case of nested or interleaved statements running under statement snapshot isolation levels, each one begins a different statement snapshot with its first read or update.

Usually there is only one transaction snapshot per connection (one entry per connection in sa_snapshots with statement_level=0). However, a snapshot associated with a cursor never changes after the cursor's first fetch and a cursor opened WITH HOLD stays open through a commit or rollback. If the cursor has an associated snapshot, then the snapshot also persists. Therefore, it is possible for multiple transaction snapshots to exist for the same connection_num: one for the current transaction snapshot and one or more for old transaction snapshots that persist because of WITH HOLD cursors.

Permissions

DBA authority required

Side effects

None

See also

- "sa_transactions system procedure" on page 932
- "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*]

sa_split_list system procedure

Takes a string of values, separated by a delimiter, and returns a set of rows—one row for each value.

Syntax

sa_split_list(
str
[, *delim*
[, *maxlen* **]** **]**
)

Arguments

- ***str*** Use this LONG VARCHAR parameter to specify the string containing the values to be split, separated by *delim*.
- ***delim*** Use this optional CHAR(10) parameter to specify the delimiter used in *str* to separate values. The delimiter can be a string of any characters, up to 10 bytes. If *delim* is not specified, a comma is used by default.
- ***maxlen*** Use this optional INTEGER parameter to specify the maximum length of the returned values. For example, if *maxlen* is set to 3, the values in the result set are truncated to a length of 3 characters. If you specify 0 (the default), values can be any length.

Result set

Column name	Data type	Description
line_num	INTEGER	Sequential number for the row.
row_value	CHAR	Value from the string, truncated to *maxlen* if required.

Remarks

The sa_split_list procedure can be used within other procedures to restrict a query to the result set of sa_split_list.

Permissions

None

Side effects

None

Examples

The following call formats the string 'yellow##blue##red' so that individual items appear on separate lines in the result set.

```
CALL sa_split_list( 'yellow##blue##red', '##', 3 );
```

line_num	row_value
1	yel
2	blu
3	red

In the following example, a procedure called ProductsWithColor is created. When called, the ProductsWithColor procedure uses sa_split_list to parse the color values specified by the user, looks in the Color column of the Products table, and returns the name, description, size, and color for each product that matches one of the user-specified colors.

The result of the procedure call below is the name, description, size, and color of all products that are either blue or white.

```
CREATE PROCEDURE ProductsWithColor( IN color_list LONG VARCHAR )
BEGIN
  SELECT Name,Description,Size,Color
  FROM Products
  WHERE Color IN ( SELECT row_value FROM sa_split_list( color_list ) )
END
GO

CALL ProductsWithColor( 'white, blue' )
```

sa_statement_text system procedure

Formats a SELECT statement so that individual items appear on separate lines. This is useful when viewing long statements from the request log, in which all newline characters are removed.

Syntax

sa_statement_text(*txt* **)**

Arguments

- ***txt*** Use this LONG VARCHAR parameter to specify a SELECT statement.

Remarks

The *txt* that is entered must be a string (in single quotes) or a string expression.

Permissions

None

Side effects

None

See also

- “sa_get_request_times system procedure” on page 873
- “sa_get_request_profile system procedure” on page 872

Example

The following call formats a SELECT statement so that individual items appear on separate lines.

```
CALL sa_statement_text( 'SELECT * FROM car WHERE name=''Audi''' );
```

	stmt_text
1	select *
2	FROM car
3	WHERE name = 'Audi'

sa_table_fragmentation system procedure

Reports information about the fragmentation of database tables.

Syntax

sa_table_fragmentation(
[*tbl_name*
[**,** *owner_name*]]
)

Arguments

- ***tbl_name*** Use this optional CHAR(128) parameter to specify the name of the table to check for fragmentation.
- ***owner_name*** Use this optional CHAR(128) parameter to specify the owner of *tbl_name*.

Result set

Column name	Data type	Description
TableName	CHAR(128)	Name of the table.
rows	UNSIGNED INTEGER	Number of rows in the table.
row_segments	UNSIGNED BIGINT	Number of row segments in the table.
segs_per_row	DOUBLE	Number of segments per row.

Remarks

Database administrators can use this procedure to obtain information about the fragmentation in a database's tables. If no arguments are supplied, results are returned for all tables in the database.

When database tables become excessively fragmented, you can run REORGANIZE TABLE or rebuild the database to reduce table fragmentation and improve performance. See "Reducing table fragmentation" [*SQL Anywhere Server - SQL Usage*].

Permissions

DBA authority required

Side effects

None

See also

- "Reducing table fragmentation" [*SQL Anywhere Server - SQL Usage*]
- "Rebuilding databases" [*SQL Anywhere Server - SQL Usage*]
- "REORGANIZE TABLE statement" on page 628

sa_table_page_usage system procedure

Reports information about the page usage of database tables.

Syntax

sa_table_page_usage( )

Result set

Column name	Data type	Description
TableId	UNSIGNED INTEGER	The table ID.
TablePages	INTEGER	The number of table pages used by the table.
PctUsedT	INTEGER	The percentage of used table page space.
IndexPages	INTEGER	The number of index pages used by the table.
PctUsedI	INTEGER	The percentage of used index page space.
PctOfFile	INTEGER	The percentage of the total database file the table occupies.
TableName	CHAR(128)	The table name.

Remarks

The results include the same information provided by the Information utility.

Permissions

DBA authority required

Side effects

None

See also

- "Information utility (dbinfo)" [*SQL Anywhere Server - Database Administration*]

sa_table_stats system procedure

Reports information about how many pages have been read from each table.

Syntax

sa_table_stats( )

Result set

Column name	Data type	Description
table_id	INT	The table ID.
creator	CHAR(128)	The user name of the table's creator.
table_name	CHAR(128)	The table name.
count	UNSIGNED BIGINT	The estimated number of rows in the table, taken from SYSTAB.
table_page_count	UNSIGNED BIGINT	The number of main pages used by the table.
table_page_cached	UNSIGNED BIGINT	The number of tables pages currently stored in the cache.
table_page_reads	UNSIGNED BIGINT	The number of page reads performed for pages in the main table.
ext_page_count	UNSIGNED BIGINT	The estimated number of pages in the table
ext_page_cached	UNSIGNED BIGINT	Reserved for future use.
ext_page_reads	UNSIGNED BIGINT	Reserved for future use.

Remarks

Each row returned by the sa_table_stats procedure describes a table for which the optimizer is maintaining page statistics. The sa_table_stats procedure can be used to find which tables are using cache memory and how many disk reads are being performed for each table. For example, you can use the sa_table_stats procedure to find the table that is generating the most disk reads. The results of the procedure represent estimates and should be used only for diagnostic purposes.

The table_page_cached column indicates how many pages of the table are currently stored in the cache, and the table_page_reads column indicates how many table pages have been read from disk since the optimizer started maintaining counts for the table. These statistics are not stored persistently within the database; they represent the activity on tables after they are loaded into memory for the first time.

Permissions

DBA authority required.

Side effects

None

See also

- "SYSTAB system view" on page 794

sa_transactions system procedure

Returns a list of transactions that are currently active.

Syntax

sa_transactions( )

Result set

Column name	Data type	Description
connection_num	INT	The connection ID for the connection the transaction is running on.
transaction _id	INT	The ID that uniquely identifies the transaction as long as the database server keeps track of it. IDs are reused as old transaction information is discarded.
start_time	TIMESTAMP	The TIMESTAMP for when the transaction started.
start_sequence_num	UNSIGNED BIG-INT	The start sequence number for the transaction.
end_sequence_num	UNSIGNED BIG-INT	Then end sequence number for the transaction if it has been committed or rolled back, otherwise, NULL.
committed	bit	The state of the transaction: true if the transaction ended with a COMMIT, false if it ended with a ROLLBACK, and NULL if the transaction is still active.
version_entries	unsigned INT	The count of the number of row versions the transaction has saved.

Remarks

This procedure provides information about the transactions that are currently running against the database.

Permissions

DBA authority required

Side effects

None

See also

- "sa_snapshots system procedure" on page 926
- "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*]

sa_unload_cost_model system procedure

Unloads the current cost model to the specified file.

Syntax

sa_unload_cost_model (*file_name* **)**

Arguments

- ***file_name*** Use this CHAR(256) parameter to specify the name of the file in which to unload the data. Because it is the database server that executes the system procedure, *file_name* specifies a file on the database server computer, and a relative *file_name* specifies a file relative to the database server's starting directory.

Remarks

The optimizer uses cost models to determine optimal access plans for queries. The database server maintains a cost model for each database. The cost model for a database can be recalibrated at any time using the CALIBRATE SERVER clause of the ALTER DATABASE statement. For example, you might decide to recalibrate the cost model if you move the database onto non-standard hardware.

The sa_unload_cost_model system procedure allows you save a cost model to an ASCII file (*file_name*). You can then log into another database and use the sa_load_cost_model system procedure to load the cost model from the first database into the second one. This avoids having to recalibrate the second database.

Note
The sa_unload_cost_model system procedure does not include CALIBRATE PARALLEL READ information in the file.

Using the sa_unload_cost_model system procedure eliminates repetitive, time-consuming recalibration activities when there is a large number of similar hardware installations.

Permissions

DBA authority required

You must have write permissions where the file is created.

Side effects

None.

See also

- "ALTER DATABASE statement" on page 301
- "sa_load_cost_model system procedure" on page 881
- "Query Optimization and Execution" [*SQL Anywhere Server - SQL Usage*]

Example

The following example unloads the cost model to a file called costmodel8:

```
CALL sa_unload_cost_model( 'costmodel8' );
```

sa_validate system procedure

Validates all tables in a database.

Syntax

sa_validate(
 [*tbl_name*
 [, *owner_name*
 [, *check_type*]]]
)

Arguments

- ***tbl_name*** Use this optional VARCHAR(128) parameter to specify the name of the table to validate. If this parameter is NULL (the default), sa_validate validates all tables.
- ***owner_name*** Use this optional VARCHAR(128) parameter to specify the owner of *tbl_name*. If this parameter is NULL (the default), sa_validate validates tables for all users.
- ***check_type*** Use this optional CHAR(10) parameter to specify the type of validation to perform. If this parameter is NULL (the default), each table is checked using a VALIDATE TABLE statement with no additional checks. The *check_type* value can be one of the following:
  - **express** Validate tables using WITH EXPRESS CHECK.
  - **checksum** Validate database pages using checksums. See "Ensuring your database is valid" [*SQL Anywhere Server - Database Administration*].

Permissions

DBA authority required

Side effects

None

Remarks

This procedure is equivalent to calling the VALIDATE TABLE statement for each table in the database. See "VALIDATE statement" on page 713.

All of the values for the *tbl_name*, *owner_name*, and *check_type* arguments are strings and they must be enclosed in quotes.

The procedure returns a single column, named Messages. If all tables are valid, the column contains `No error detected`.

Caution
Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, spurious errors may be reported indicating some form of database corruption even though no corruption actually exists.

Example

The following statement performs an express check on tables owned by DBA:

```
CALL sa_validate( owner_name = 'DBA', check_type = 'checksum' );
```


sa_verify_password system procedure

Validates the password of the current user.

Syntax

sa_verify_password(*curr_pwsd* **)**

Arguments

- ***curr_pwsd*** Use this CHAR(128) parameter to specify the password of the current database user.

Remarks

This procedure is used by sp_password. If the password matches, the procedure simply returns. If it does not match, the error `Invalid user ID or password` is returned.

Permissions

None

Side effects

None

See also

- "Adaptive Server Enterprise system procedures" on page 962

sp_login_environment system procedure

Sets connection options when users log in.

Syntax

sp_login_environment( )

Remarks

sp_login_environment is the default procedure called by the login_procedure database option.

It is recommended that you do not edit this procedure. Instead, to change the login environment, set the login_procedure option to point to a different procedure.

Here is the text of the sp_login_environment procedure:

```
CREATE PROCEDURE dbo.sp_login_environment( )
BEGIN
  IF connection_property( 'CommProtocol' ) = 'TDS' THEN
    CALL dbo.sp_tsql_environment( )
  END IF
END;
```

Permissions

None

Side effects

None

See also

- "login_procedure option [database]" [*SQL Anywhere Server - Database Administration*]

sp_remote_columns system procedure

Produces a list of the columns in a remote table, and a description of their data types.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_columns(
@server_name,
@table_name
[, *@table_owner*
[, *@table_qualifier*]]
)

Arguments

- ***@server_name*** Use this CHAR(128) parameter to specify a string containing the server name as specified by the CREATE SERVER statement.
- ***@table_name*** Use this CHAR(128) parameter to specify the name of the remote table.
- ***@table_owner*** Use this optional CHAR(128) parameter to specify the owner of *@table_name*.
- ***@table_qualifier*** Use this optional CHAR(128) parameter to specify the name of the database in which *@table_name* is located.

Result set

Column name	Data type	Description
database	CHAR(128)	The database name.
owner	CHAR(128)	The database owner name.
table-name	CHAR(128)	The table name.
column-name	CHAR(128)	The name of a column.
domain-id	SMALLINT	An INTEGER which indicates the data type of the column.
width	SMALLINT	The meaning of this field depends on the data type. For character types width represents the number of characters.
scale	SMALLINT	The meaning of this field depends on the data type. For NUMERIC data types scale is the number of digits after the decimal point.

Column name	Data type	Description
nullable	SMALLINT	If null column values are allowed this field is 1. Otherwise nullable is 0.

Remarks

If you are entering a CREATE EXISTING statement and you are specifying a column list, it may be helpful to get a list of the columns that are available on a remote table. sp_remote_columns produces a list of the columns on a remote table and a description of their data types. If you specify a database, you must either specify an owner or provide the value **null**.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "Accessing Remote Data" [*SQL Anywhere Server - SQL Usage*]
- "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*]
- "CREATE SERVER statement" on page 435

Example

The following example returns columns from the SYSOBJECTS table in the production database on an Adaptive Server Enterprise server named asetest. The owner is unspecified.

```
CALL sp_remote_columns( 'asetest', 'sysobjects', null, 'production' );
```

sp_remote_exported_keys system procedure

Provides information about tables with foreign keys on a specified primary table.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_exported_keys(
@server_name
, *@sp_name*
[**,** *@sp_owner*
[**,** *@sp_qualifier*]]
)

Arguments

- ***@server_name*** Use this CHAR(128) parameter to specify identifies the server the primary table is located on. A value is required for this parameter.

- ***@sp_name*** Use this CHAR(128) parameter to specify the table containing the primary key. A value is required for this parameter.
- ***@sp_owner*** Use this optional CHAR(128) parameter to specify the primary table's owner.
- ***@sp_qualifier*** Use this optional CHAR(128) parameter to specify the database containing the primary table.

Result set

Column name	Data type	Description
pk_database	CHAR(128)	The database containing the primary key table.
pk_owner	CHAR(128)	The owner of the primary key table.
pk_table	CHAR(128)	The primary key table.
pk_column	CHAR(128)	The name of the primary key column.
fk_database	CHAR(128)	The database containing the foreign key table.
fk_owner	CHAR(128)	The foreign key table's owner.
fk_table	CHAR(128)	The foreign key table.
fk_column	CHAR(128)	The name of the foreign key column.
key_seq	SMALLINT	The key sequence number.
fk_name	CHAR(128)	The foreign key name.
pk_name	CHAR(128)	The primary key name.

Remarks

This procedure provides information about the remote tables that have a foreign key on a particular primary table. The result set for the sp_remote_exported_keys system procedure includes the database, owner, table, column, and name for both the primary and the foreign key, as well as the foreign key sequence for the foreign key columns. The result set may vary because of the underlying ODBC and JDBC calls, but information about the table and column for a foreign key is always returned.

Permissions

None

Side effects

None

See also

- "CREATE SERVER statement" on page 435
- "Foreign keys" [*SQL Anywhere 10 - Introduction*]

Example

To get information about the remote tables with foreign keys on the SYSOBJECTS table, in the production database, on a server named asetest:

```
CALL sp_remote_exported_keys(
     @server_name='asetest',
     @sp_name='sysobjects',
     @sp_qualifier='production' )
```

sp_remote_imported_keys system procedure

Provides information about remote tables with primary keys that correspond to a specified foreign key.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_imported_keys(
@server_name
, *@sp_name*
[, *@sp_owner*
[, *@sp_qualifier*]]
)

Arguments

- ***@server_name*** Use this optional CHAR(128) parameter to specify the server the foreign key table is located on. A value is required for this parameter.
- ***@sp_name*** Use this optional CHAR(128) parameter to specify the table containing the foreign key. A value is required for this parameter.
- ***@sp_owner*** Use this optional CHAR(128) parameter to specify the foreign key table's owner.
- ***@sp_qualifier*** Use this optional CHAR(128) parameter to specify the database containing the foreign key table.

Result set

Column name	Data type	Description
pk_database	CHAR(128)	The database containing the primary key table.
pk_owner	CHAR(128)	The owner of the primary key table.
pk_table	CHAR(128)	The primary key table.
pk_column	CHAR(128)	The name of the primary key column.
fk_database	CHAR(128)	The database containing the foreign key table.
fk_owner	CHAR(128)	The foreign key table's owner.
fk_table	CHAR(128)	The foreign key table.

Column name	Data type	Description
fk_column	CHAR(128)	The name of the foreign key column.
key_seq	SMALLINT	The key sequence number.
fk_name	CHAR(128)	The foreign key name.
pk_name	CHAR(128)	The primary key name.

Remarks

Foreign keys reference a row in a separate table that contains the corresponding primary key. This procedure allows you to obtain a list of the remote tables with primary keys that correspond to a particular foreign table. The sp_remote_imported_keys result set includes the database, owner, table, column, and name for both the primary and the foreign key, as well as the foreign key sequence for the foreign key columns. The result set may vary because of the underlying ODBC and JDBC calls, but information about the table and column for a primary key is always returned.

Permissions

None

Side effects

None

See also

- "CREATE SERVER statement" on page 435
- "Foreign keys" [*SQL Anywhere 10 - Introduction*]

Example

To get information about the tables with primary keys that correspond to a foreign key on the SYSOBJECTS table in the asetest server:

```
CALL sp_remote_imported_keys(
     @server_name='asetest',
     @sp_name='sysobjects',
     @sp_qualifier='production' );
```

sp_remote_primary_keys system procedure

Provides primary key information about remote tables using remote data access.

Syntax

sp_remote_primary_keys(
@server_name
[, *@table_name*
[, *@table_owner*
[, *@table_qualifier*]]]
)

Arguments

- ***@server_name*** Use this CHAR(128) parameter to specify the server the remote table is located on.
- ***@table_name*** Use this optional CHAR(128) parameter to specify the remote table.
- ***@table_owner*** Use this optional CHAR(128) parameter to specify the owner of the remote table.
- ***@table_qualifier*** Use this optional CHAR(128) parameter to specify the name of the remote database.

Result set

Column name	Data type	Description
database	CHAR(128)	The name of the remote database.
owner	CHAR(128)	The owner of the remote table.
table-name	CHAR(128)	The remote table.
column-name	CHAR(128)	The column name.
key-seq	SMALLINT	The primary key sequence number.
pk-name	CHAR(128)	The primary key name.

Remarks

This system procedure provides primary key information about remote tables using remote data access.

Because of differences in the underlying ODBC/JDBC calls, the information returned differs slightly in terms of the catalog/database value depending upon the remote data access class that is specified for the server. However, the important information (for example, column name) is as expected.

Standards and compatibility

Sybase Supported by Open Client/Open Server.

Permissions

None

Side effects

None

sp_remote_tables system procedure

Returns a list of the tables on a server.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_tables(
@server_name

```
 [, @table_name
 [, @table_owner
 [, @table_qualifier
 [, @with_table_type ] ] ] ]
)
```

Arguments

- ***@server_name*** Use this CHAR(128) parameter to specify the server the remote table is located on.
- ***@table_name*** Use this CHAR(128) parameter to specify the remote table.
- ***@table_owner*** Use this CHAR(128) parameter to specify the owner of the remote table.
- ***@table_qualifier*** Use this CHAR(128) parameter to specify the database in which *table_name* is located.
- ***@with_table_type*** Use this optional BIT parameter to specify the type of remote table. This argument is a bit type and accepts two values, 0 (the default) and 1. You must enter the value 1 if you want the result set to include a column that lists table types.

Result set

Column name	Data type	Description
database	CHAR(128)	The name of the remote database.
owner	CHAR(128)	The name of the remote database owner.
table-name	CHAR(128)	The remote table.
table-type	CHAR(128)	Specifies the table type. The value of this field depends on the type of remote server. For example, TABLE, VIEW, SYS, and GBL TEMP are possible values.

Remarks

It may be helpful when you are configuring your database server to get a list of the remote tables available on a particular server. This procedure returns a list of the tables on a server.

The procedure accepts five parameters. If a table, owner, or database name is given, the list of tables will be limited to only those that match the arguments.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "Accessing Remote Data" [*SQL Anywhere Server - SQL Usage*]
- "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*]
- "CREATE SERVER statement" on page 435

Examples

To get a list of all of the Microsoft Excel worksheets available from an ODBC data source referenced by a server named excel:

```
CALL sp_remote_tables( 'excel' );
```

To get a list of all of the tables owned by fred in the production database in an Adaptive Server Enterprise server named asetest:

```
CALL sp_remote_tables( 'asetest', null, 'fred', 'production' );
```

sp_servercaps system procedure

Displays information about a remote server's capabilities.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_servercaps(*@sname* **)**

Arguments

- ***@sname*** Use this CHAR(64) parameter to specify a server defined with the CREATE SERVER statement. The specified *@sname* must be the same server name used in the CREATE SERVER statement.

Remarks

This procedure displays information about a remote server's capabilities. SQL Anywhere uses this capability information to determine how much of a SQL statement can be forwarded to a remote server. The system tables that contain server capabilities (ISYSCAPABILITY and ISYSCAPABILITYNAME) are not populated until after SQL Anywhere first connects to the remote server.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "SYSCAPABILITY system view" on page 755
- "SYSCAPABILITYNAME system view" on page 756

- "Accessing Remote Data" [*SQL Anywhere Server - SQL Usage*]
- "Server Classes for Remote Data Access" [*SQL Anywhere Server - SQL Usage*]
- "CREATE SERVER statement" on page 435

Example

To display information about the remote server testasa:

```
CALL sp_servercaps( 'testasa' );
```

sp_tsql_environment system procedure

Sets connection options when users connect from jConnect or Open Client applications.

Syntax

sp_tsql_environment( )

Remarks

The sp_login_environment procedure is the default procedure specified by the login_procedure database option. For each new connection, the procedure specified by login_procedure is called. If the connection uses the TDS communications protocol (that is, if it is an Open Client or jConnect connection), then sp_login_environment in turn calls sp_tsql_environment.

This procedure sets database options so that they are compatible with default Sybase Adaptive Server Enterprise behavior.

If you want to change the default behavior, it is recommended that you create new procedures and alter your login_procedure option to point to these new procedures.

Permissions

None

Side effects

None

See also

- "sp_login_environment system procedure" on page 936
- "login_procedure option [database]" [*SQL Anywhere Server - Database Administration*].

Example

Here is the text of the sp_tsql_environment procedure:

```
CREATE PROCEDURE dbo.sp_tsql_environment( )
BEGIN
  IF db_property( 'IQStore' )='OFF' THEN
    -- SQL Anywhere datastore
    SET TEMPORARY OPTION automatic_TIMESTAMP='On'
  END IF;
  SET TEMPORARY OPTION ansinull='Off';
  SET TEMPORARY OPTION tsql_variables='On';
  SET TEMPORARY OPTION ansi_blanks='On';
```



```
    SET TEMPORARY OPTION tsql_hex_constant='On';
    SET TEMPORARY OPTION chained='Off';
    SET TEMPORARY OPTION quoted_identifier='Off';
    SET TEMPORARY OPTION allow_nulls_by_default='Off';
    SET TEMPORARY OPTION float_as_double='On';
    SET TEMPORARY OPTION on_tsql_error='Continue';
    SET TEMPORARY OPTION isolation_level='1';
    SET TEMPORARY OPTION date_format='YYYY-MM-DD';
    SET TEMPORARY OPTION TIMESTAMP_format='YYYY-MM-DD HH:NN:SS.SSS';
    SET TEMPORARY OPTION time_format='HH:NN:SS.SSS';
    SET TEMPORARY OPTION date_order='MDY';
    SET TEMPORARY OPTION escape_character='Off'
    SET TEMPORARY OPTION close_on_endtrans='Off'
END;
```

xp_cmdshell system procedure

Carries out an operating system command from a procedure.

Syntax

xp_cmdshell(
command
[, **'no_output'**] **)**

Arguments

- ***command*** Use this CHAR(8000) parameter to specify a system command.
- **'no_output'** Use this optional CHAR(254) parameter to specify whether to display output. The default behavior is to display output. If this parameter is the string **'no_output'**, no output is displayed.

Remarks

xp_cmdshell executes a system command and then returns control to the calling environment.

The second parameter affects only console applications on Windows operating systems. For Unix, no output is displayed regardless of the setting for the second parameter. For NetWare, any commands executed are visible on the server console, regardless of the setting for the second parameter.

For NetWare and Windows CE, any commands executed are visible on the server console, regardless of the setting for the second parameter. On Windows CE, the console shell *\\windows\cmd.exe* is needed to run the procedure.

Permissions

DBA authority required

See also

- "CALL statement" on page 357

Example

The following statement lists the files in the current directory in the file *c:\temp.txt*:

```
xp_cmdshell( 'dir > c:\\temp.txt' )
```


The following statement carries out the same operation, but does so without displaying a command window.

```
xp_cmdshell( 'dir > c:\\temp.txt', 'no_output' )
```

xp_read_file system procedure

Returns the contents of a file as a LONG BINARY variable.

Syntax

xp_read_file(*filename* **)**

Arguments

- ***filename*** Use this LONG VARCHAR parameter to specify the name of the file for which to return the contents.

Remarks

The function reads the contents of the named file, and returns the result as a LONG BINARY value.

The *filename* is relative to the starting directory of the database server.

The function can be useful for inserting entire documents or images stored in files into tables. If the file cannot be read, the function returns NULL.

Permissions

DBA authority required

See also

- "xp_write_file system procedure" on page 949
- "CALL statement" on page 357
- "Using openxml with xp_read_file" [*SQL Anywhere Server - SQL Usage*]

Example

The following statement inserts an image into a column named picture of the table t1 (assuming all other columns can accept NULL):

```
INSERT INTO t1 ( picture )
   SELECT xp_read_file( 'portrait.gif' );
```

xp_scanf system procedure

Extracts substrings from an input string and a format string.

Syntax

xp_scanf(
input_buffer,
format,
parm [**,** *parm2*, ...]
)

Arguments

- ***input_buffer*** Use this CHAR(254) parameter to specify the input string.
- ***format*** Use this CHAR(254) parameter to specify the format of the input string, using placeholders (% s) for each *parm* argument. There can be up to fifty placeholders in the *format* argument, and there must be the same number of placeholders as *parm* arguments.
- ***parm*** Use one or more of these CHAR(254) parameters to specify the substrings extracted from *input_buffer*. There can be up to 50 of these parameters.

Remarks

The xp_scanf system procedure extracts substrings from an input string using the specified *format*, and puts the results in the specified *parm* values.

Permissions

None

See also

- "CALL statement" on page 357

Example

The following statements extract the substrings Hello and World! from the input buffer Hello World!, and put them into variables string1 and string2, and then selects them:

```
CREATE VARIABLE string1 CHAR(254);
CREATE VARIABLE string2 CHAR(254);
CALL xp_scanf( 'Hello World!', '%s %s', string1, string2 );
SELECT string1, string2;
```

xp_sprintf system procedure

Builds a result string from a set of input strings.

Syntax

xp_sprintf(
output_buffer
, *format*
, *parm* [**,** *parm2*, ...]
)

Arguments

- ***output_buffer*** Use this CHAR(254) parameter to specify the output buffer containing the result string.
- ***format*** Use this CHAR(254) parameter to specify how to format the result string, using placeholders (%s) for each *parm* argument. There can be up to fifty placeholders in the *format* argument, and there should be the same number of placeholders as *parm* arguments.
- ***parm*** These are the input strings that are used in the result string. You can specify up to 50 of these CHAR(254) arguments.

Remarks

The xp_sprintf system procedure builds up a string using the *format* argument and the *parm* argument(s), and puts the results in *output_buffer*.

Permissions

None

See also

- "CALL statement" on page 357

Example

The following statements put the string Hello World! into the result variable.

```
CREATE VARIABLE result CHAR(254);
Call xp_sprintf( result, '%s %s', 'Hello', 'World!' );
```

xp_write_file system procedure

Writes data to a file from a SQL statement.

Syntax

xp_write_file(
filename
, *file_contents*
)

Arguments

- ***filename*** Use this LONG VARCHAR parameter to specify the file name.
- ***file_contents*** Use this LONG BINARY parameter to specify the contents to write to the file.

Remarks

The function writes *file_contents* to the file *filename*. It returns 0 if successful, and non-zero if it fails.

The *filename* value can be prefixed by either an absolute or a relative path. If *filename* is prefixed by a relative path, then the file name is relative to the current working directory of the database server. If the file already exists, its contents are overwritten.

This function can be useful for unloading long binary data into files.

Permissions

DBA authority required

See also

- "xp_read_file system procedure" on page 947
- "CALL statement" on page 357

Examples

This example uses xp_write_file to create a file *accountnum.txt* containing the data 123456:

```
CALL  xp_write_file( 'accountnum.txt', '123456' );
```

This example queries the Contacts table of the sample database, and then creates a text file for each contact living in New Jersey. Each text file is named using a concatenation of the contact's first name (GivenName), last name (Surname), and then the string .txt (for example, *Reeves_Scott.txt*), and contains the contact's street address (Street), city (City), and state (State), on separate lines.

```
SELECT xp_write_file(
Surname || '_' ||  GivenName || '.txt',
Street || '\n' || City || '\n' || State )
FROM Contacts WHERE State = NJ;
```

This example uses xp_write_file to create an image file (JPG) for every product in the Products table. Each value of the ID column becomes a file name for a file with the contents of the corresponding value of the Photo column:

```
SELECT xp_write_file( ID || '.jpg' , Photo ) FROM Products;
```

In the example above, ID is a row with a UNIQUE constraint. This is important to ensure that a file isn't overwritten with the contents of subsequent row. Also, you must specify the file extension applicable to the data stored in the column. In this case, the Products.Photo stores image data (JPGs).

System extended procedures

A set of system extended procedures are included in SQL Anywhere databases. These procedures are owned by the dbo user ID. Users must be granted EXECUTE permission before they can use these procedures, unless they already have DBA authority.

Extended system procedures for MAPI and SMTP

SQL Anywhere includes system procedures for sending electronic mail using the Microsoft Messaging API standard (MAPI) or the Internet standard Simple Mail Transfer Protocol (SMTP). These system procedures are implemented as extended system procedures: each procedure calls a function in an external DLL.

To use the MAPI or SMTP system procedures, a MAPI or SMTP email system must be accessible from the database server computer.

The system procedures are:

- **xp_startmail** Starts a mail session in a specified mail account by logging onto the MAPI message system
- **xp_startsmtp** Starts a mail session in a specified mail account by logging onto the SMTP message system
- **xp_sendmail** Sends a mail message to specified users
- **xp_stopmail** Closes the MAPI mail session
- **xp_stopsmtp** Closes the SMTP mail session

The following procedure notifies a set of people that a backup has been completed.

```
CREATE PROCEDURE notify_backup( )
BEGIN
   CALL xp_startmail( mail_user='ServerAccount',
                  mail_password='ServerPassword'
                     );
   CALL xp_sendmail( recipient='IS Group',
                     subject='Backup',
                     "message"='Backup completed'
                     );
   CALL xp_stopmail( )
END
```

The mail system procedures are discussed in the following sections.

xp_startmail system procedure

Starts an email session under MAPI.

Syntax

xp_startmail(
[**mail_user** = *mail-login-name*]
[**, mail_password** = *mail-password*] **)**

Arguments

- **mail_user** Use this LONG VARCHAR parameter to specify the MAPI login name.
- **mail_password** Use this LONG VARCHAR parameter to specify the MAPI password.

Permissions

DBA authority required

Not supported on NetWare or Unix.

Remarks

xp_startmail is a system procedure that starts an email session.

If you are using Microsoft Exchange, the *mail-login-name* argument is an Exchange profile name, and you should not include a password in the procedure call.

Return codes

See "MAPI return codes" on page 958.

See also

- "CALL statement" on page 357

xp_startsmtp system procedure

Starts an email session under SMTP.

Syntax

xp_startsmtp(
smtp_sender = = *email-address***,**
smtp_server = *smtp-server*
[**, smtp_port =** *port-number*]
[**, timeout =** *timeout*]
[**, smtp_sender_name =** *username*]
[**, smtp_auth_username =** *auth-username*
[**, smtp_auth_password =** *auth-password*
)

Arguments

- **smtp_sender** This LONG VARCHAR parameter specifies the email address of the sender.
- **smtp_server** This LONG VARCHAR parameter specifies which SMTP server to use, and is the server name or IP address.
- **smtp_port** This optional INTEGER parameter specifies the port number to connect to on the SMTP server. The default is 25.

- **timeout** This optional INTEGER parameter specifies how long to wait, in seconds, for a response from the database server before aborting the current call to xp_sendmail. The default is 60 seconds.
- **smtp_sender_name** This optional LONG VARCHAR parameter specifies an alias for the sender's email address. For example, 'JSmith' instead of '*email-address*'.
- **smtp_auth_username** This optional LONG VARCHAR parameter specifies the user name to provide to SMTP servers requiring authentication.
- **smtp_auth_password** This optional LONG VARCHAR parameter specifies the user name to provide to SMTP servers requiring authentication.

Permissions

DBA authority required

Not supported on NetWare.

Remarks

xp_startsmtp is a system procedure that starts a mail session for a specified email address by connecting to an SMTP server. This connection can time out. Therefore, it is recommended that you call xp_start_smtp just before executing xp_sendmail.

Virus scanners can affect xp_startsmtp, causing it to return error code 100. For McAfee VirusScan version 8.0.0 and higher, settings for preventing mass mailing of email worms also prevent xp_sendmail from executing properly. If your virus scanning software allows you to specify processes that can bypass the mass mailing protections, specify *dbeng10.exe* and *dbsvr10.exe*. For example, in the case of McAfee VirusScan, you can add these two processes to the list of Excluded Processes in the Properties area for preventing mass mailing.

Return codes

See "SMTP return codes" on page 959.

See also

- "CALL statement" on page 357
- "xp_startsmtp system procedure" on page 952
- "xp_stopsmtp system procedure" on page 957

xp_sendmail system procedure

Sends an email message.

Syntax

xp_sendmail(
recipient = *mail-address*
[**, subject** = *subject*]
[**, cc_recipient** = *mail-address*]
[**, bcc_recipient** = *mail-address*]
[**, query** = *sql-query*]
[**, "message"** = *message-body*]

[, **attachname** = *attach-name*]
[, **attach_result** = *attach-result*]
[, **echo_error** = *echo-error*]
[, **include_file** = *file-name*]
[, **no_column_header** = *no-column-header*]
[, **no_output** = *no-output*]
[, **width** = *width*]
[, **separator** = *separator-char*]
[, **dbuser** = *user-name*]
[, **dbname** = *db-name*]
[, **type** = *type*]
[, **include_query** = *include-query*]
[, **content_type** = *content-type*]
)

Arguments

Some arguments supply fixed values and are available for use to ensure Transact-SQL compatibility, as noted below.

- **recipient** This LONG VARCHAR parameter specifies the recipient mail address. When specifying multiple recipients, each mail address must be separated by a semicolon.
- **subject** This LONG VARCHAR parameter specifies the subject field of the message. The default is NULL.
- **cc_recipient** This LONG VARCHAR parameter specifies the cc recipient mail address. When specifying multiple cc recipients, each mail address must be separated by a semicolon. The default is NULL.
- **bcc_recipient** This LONG VARCHAR parameter specifies the bcc recipient mail address. When specifying multiple bcc recipients, each mail address must be separated by a semicolon. The default is NULL.
- **query** This LONG VARCHAR is for use with Transact-SQL. The default is NULL.
- **"message"** This LONG VARCHAR parameter specifies the message contents. The default is NULL. The "message" parameter name requires double quotes around it because MESSAGE is a reserved word. See “Reserved words” on page 4.
- **attachname** This LONG VARCHAR parameter is for use with Transact-SQL. The default is NULL.
- **attach_result** This INT parameter is for use with Transact-SQL. The default is 0.
- **echo_error** This INT parameter is for use with Transact-SQL. The default is 1.
- **include_file** This LONG VARCHAR parameter specifies an attachment file. The default is NULL.
- **no_column_header** This INT parameter is for use with Transact-SQL. The default is 0.
- **no_output** This INT parameter is for use with Transact-SQL. The default is 0.
- **width** This INT parameter is for use with Transact-SQL. The default is 80.
- **separator** This CHAR(1) parameter is for use with Transact-SQL. The default is CHAR(9).

- **dbuser** This LONG VARCHAR parameter is for use with Transact-SQL. The default is guest.
- **dbname** This LONG VARCHAR parameter is for use with Transact-SQL. The default is master.
- **type** This LONG VARCHAR parameter is for use with Transact-SQL. The default is NULL.
- **include_query** This INT parameter is for use with Transact-SQL. The default is 0.
- **content_type** This LONG VARCHAR parameter specifies the content type for the "message" parameter (for example, text/html, ASIS, and so on). The default is NULL. The value of content_type is not validated; setting an invalid content type results in an invalid or incomprehensible email being sent.

Permissions

DBA authority required

Must have executed xp_startmail to start an email session using MAPI, or xp_startsmtp to start an email session using SMTP.

If you are sending mail using MAPI, the content_type parameter is not supported.

Not supported on NetWare.

Remarks

xp_sendmail is a system procedure that sends an email message to the specified recipients once a session has been started with xp_startmail or xp_startsmtp. The procedure accepts messages of any length. The argument values for xp_sendmail are strings. The length of each argument is limited to the amount of available memory on your system.

The content_type argument is intended for users who understand the requirements of MIME email. xp_sendmail accepts ASIS as a content_type. When content_type is set to ASIS, xp_sendmail assumes that the message body ("message") is already a properly formed email with headers, and does not add any additional headers. Specify ASIS to send multipart messages containing more than one content type. See RFCs 2045-2049 (http://www.ietf.org/) for more information on MIME.

Attachments specified by the include_file parameter are sent as application/octet-stream MIME type, with base64 encoding, and must be present on the database server.

Return codes

See “Return codes for MAPI and SMTP system procedures” on page 958.

See also

- “CALL statement” on page 357
- “xp_startmail system procedure” on page 951
- “xp_startsmtp system procedure” on page 952
- “xp_stopmail system procedure” on page 957
- “xp_stopsmtp system procedure” on page 957

Example

The following call sends a message to the user ID Sales Group containing the file *prices.doc* as a mail attachment:


```
CALL xp_sendmail( recipient='Sales Group',
      subject='New Pricing',
      include_file = 'C:\\DOCS\\PRICES.DOC' )
```

The following sample program shows various uses of the xp_sendmail system procedure, as described in the example itself:

```
BEGIN
DECLARE to_list LONG VARCHAR;
DECLARE email_subject CHAR(256);
DECLARE content LONG VARCHAR;
DECLARE uid CHAR(20);

set to_list='test_account@mytestdomain.com';
set email_subject='This is a test';
set uid='test_sender@mytestdomain.com';

// Call xp_startsmtp to start an SMTP email session
call xp_startsmtp( uid, 'mymailserver.mytestdomain.com' );

// Basic email example
set content='This text is the body of my email.\n';
call xp_sendmail( recipient=to_list,
     subject=email_subject,
     "message"=content );

// Send email containing HTML using the content_type parameter,
// as well as including an attachment with the include_file
// parameter
set content='Plain text.<BR><BR><B>Bold text.</B><BR><BR><a
href="www.iAnywhere.com">iAnywhere
 Home Page</a></B><BR><BR>';
call xp_sendmail( recipient=to_list,
     subject=email_subject,
     "message"=content,
     content_type = 'text/html',
     include_file = 'test.zip' );

// Send email "ASIS".  Here the content-type has been specified
// by the user as part of email body.  Note the attachment can
// also be done separately
set content='Content-Type: text/html;\nContent-Disposition: inline; \n\nThis
text
 is not bold<BR><BR><B>This text is bold</B><BR><BR><a
href="www.iAnywhere.com">iAnywhere Home
 Page</a></B><BR><BR>';
call xp_sendmail( recipient=to_list,
     subject=email_subject,
     "message"=content,
     content_type = 'ASIS',
     include_file = 'test.zip' );

// Send email "ASIS" along with an include file.  Note that
// "message" contains the information for another attachment
set content = 'Content-Type: multipart/mixed; boundary="xxxxx";\n';
set content = content || 'This part of the email should not be shown.  If this
is shown
 then the email client is not MIME compatible\n\n';
set content = content || '--xxxxx\n';
set content = content || 'Content-Type: text/html;\n';
set content = content || 'Content-Disposition: inline;\n\n';
set content = content || 'This text is not bold<BR><BR><B>This text is bold</
B><BR>
```



```
 <BR><a href="www.iAnywhere.com">iAnywhere Home Page</a></B><BR><BR>\n\n';
set content = content || '--xxxxx\n';
set content = content || 'Content-Type: application/zip; name="test.zip"\n';
set content = content || 'Content-Transfer-Encoding: base64\n';
set content = content || 'Content-Disposition: attachment;
filename="test.zip"\n\n';

// Encode the attachment yourself instead of adding this one in
// the include_file parameter
set content = content || base64_encode( xp_read_file( 'othertest.zip' ) ) ||
'\n\n';
set content = content || '--xxxxx--\n';
call xp_sendmail( recipient=to_list,
      subject=email_subject,
      "message"=content,
      content_type = 'ASIS',
      include_file = 'othertest.zip' );

// end the SMTP session
    call xp_stopsmtp();
END
```

xp_stopmail system procedure

Closes a MAPI email session.

Syntax

xp_stopmail( )

Permissions

DBA authority required

Not supported on NetWare or Unix.

Remarks

xp_stopmail is a system procedure that ends an email session.

Return codes

See “MAPI return codes” on page 958.

See also

- “CALL statement” on page 357

xp_stopsmtp system procedure

Closes an SMTP email session.

Syntax

xp_stopsmtp( )

Permissions

DBA authority required

Not supported on NetWare

Remarks

xp_stopsmtp is a system procedure that ends an email session.

Return codes

See "SMTP return codes" on page 959.

See also

- "CALL statement" on page 357

Return codes for MAPI and SMTP system procedures

Extended system procedures for MAPI and SMTP use the following return codes.

MAPI return codes

Return code	Meaning
0	Success.
2	xp_startmail failed.
3	xp_stopmail failed.
5	xp_sendmail failed.
11	Ambiguous recipient.
12	Attachment not found.
13	Disk full.
14	Failure
15	Insufficient memory.
16	Invalid session.
17	Text too large.
18	Too many files.
19	Too many recipients.
20	Unknown recipient.
21	Login failure.

Return code	Meaning
22	Too many sessions.
23	User abort.
24	No MAPI.
25	No startmail.

SMTP return codes

Return code	Meaning
0	Success.
100	Socket error.
101	Socket timeout.
102	Unable to resolve the SMTP server hostname.
103	Unable to connect to the SMTP server.
104	Server error; response not understood. For example, the message is poorly formatted, or the server is not SMTP.
421	<*domain*> service not available, closing transmission channel.
450	Requested mail action not taken: mailbox unavailable.
451	Requested action not taken: local error in processing.
452	Requested action not taken: insufficient system storage.
500	Syntax error, command unrecognized. (This may include errors such as a command that is too long).
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command parameter not implemented.
550	Requested action not taken: mailbox unavailable. For example, the mailbox is not found, there is no access, or no relay is allowed.
551	User not local; please try <*forward-path*>
552	Request mail action aborted: exceeded storage allocation.

Return code	Meaning
553	Requested action not taken: mailbox name not allowed. For example, the mailbox syntax is incorrect.
554	Transaction failed.

Other extended system procedures

Additional extended system procedures allow you to execute system commands. For example, you can use these system procedures for file I/O and string operations.

xp_msver system procedure

Retrieves version and name information about the database server.

Syntax

xp_msver(*string* **)**

- ***string*** The string must be one of the following, enclosed in string delimiters.

Argument	Description
ProductName	The name of the product (Sybase SQL Anywhere).
ProductVersion	The version number, followed by the build number. The format is as follows: `10.0.1 (30)`
CompanyName	Returns the following string: `iAnywhere Solutions, Inc.`
FileDescription	Returns the name of the product, followed by the name of the operating system.
LegalCopyright	Returns a copyright string for the software.
LegalTrademarks	Returns trademark information for the software.

Remarks

xp_msver returns product, company, version, and other information.

Permissions

None

See also

- "System functions" on page 100

Example

The following statement requests the version and operating system description:

```
SELECT xp_msver( 'ProductVersion') Version,
   xp_msver( 'FileDescription' ) Description
```

Sample output is as follows:

Version	Description
9.0.0 (1912)	Sybase SQL Anywhere Windows NT

Adaptive Server Enterprise system and catalog procedures

Adaptive Server Enterprise provides system and catalog procedures to carry out many administrative functions and to obtain system information. System procedures are built-in stored procedures used for getting reports from and updating system tables; catalog stored procedures retrieve information from the system tables in tabular form.

SQL Anywhere has implemented support for some of these Adaptive Server Enterprise procedures. However, for information on how to use these procedures, refer to your Adaptive Server Enterprise documentation.

Adaptive Server Enterprise system procedures

The following list describes the Adaptive Server Enterprise system procedures that are provided in SQL Anywhere.

While these procedures perform the same functions as they do in Adaptive Server Enterprise and pre-Version 12 Adaptive Server IQ, they are not identical. If you have preexisting scripts that use these procedures, you may want to examine the procedures. To see the text of a stored procedure, you can open it in Sybase Central or, in Interactive SQL, run the following command.

```
sp_helptext 'dbo.procedure_name'
```

You may need to reset the width of your Interactive SQL output to see the full text, by selecting Command ► Options and entering a new Limit Display Columns value.

System procedure name	Description
sp_addgroup	Adds a group to a database
sp_addlogin	Adds a new login ID to a database
sp_addmessage	Adds a user-defined message to ISYSUSERMESSAGE, for use by stored procedure PRINT and RAISERROR calls
sp_addtype	Creates a user-defined data type
sp_adduser	Adds a new user ID to a database
sp_changegroup	Changes a user's group or adds a user to a group
sp_dropgroup	Drops a group from a database
sp_droplogin	Drops a login ID from a database
sp_dropmessage	Drops a user-defined message
sp_droptype	Drops a user-defined data type

System procedure name	Description
sp_dropuser	Drops a user ID from a database
sp_getmessage	Retrieves a stored message string from ISYSUSERMESSAGE, for PRINT and RAISERROR statements.
sp_helptext	Displays the text of a system procedure, trigger, or view
sp_password	Adds or changes a password for a user ID

Adaptive Server Enterprise catalog procedures

SQL Anywhere implements a subset of the Adaptive Server Enterprise catalog procedures. The implemented catalog procedures are described in the following table.

Catalog procedure name	Description
sp_column_privileges	Unsupported
sp_columns	Returns the data types of the specified columns
sp_fkeys	Returns foreign key information about the specified table
sp_pkeys	Returns primary key information about the specified table
sp_special_columns	Returns the optimal set of columns that uniquely identify a row in the specified table
sp_sproc_columns	Returns information about a stored procedure's input and return parameters
sp_statistics	Returns information about tables and their indexes
sp_stored_procedures	Returns information about one or more stored procedures
sp_tables	Returns a list of objects that can appear in a FROM clause for the specified table

Index

Symbols

% comment indicator
about, 42
% operator
modulo function, 201
&
bitwise operator, 13
/* comment indicator
about, 42
// comment indicator
about, 42
0x
binary literals, 9
@@char_convert global variable
about, 38
@@client_csid global variable
about, 38
@@client_csname global variable
about, 38
@@connections global variable
about, 38
@@cpu_busy global variable
about, 38
@@dbts global variable
about, 38
@@error global variable
about, 38
@@fetch_status global variable
about, 38
@@identity global variable
about, 38
description, 41
triggers, 41
@@idle global variable
about, 38
@@io_busy global variable
about, 38
@@isolation global variable
about, 38
@@langid global variable
about, 38
@@language global variable
about, 38
@@max_connections global variable
about, 38
@@maxcharlen global variable
about, 38
@@ncharsize global variable
about, 38
@@nestlevel global variable
about, 38
@@pack_received global variable
about, 38
@@pack_sent global variable
about, 38
@@packet_errors global variable
about, 38
@@procid global variable
about, 38
@@rowcount global variable
about, 38
@@servername global variable
about, 38
@@spid global variable
about, 38
@@sqlstatus global variable
about, 38
@@textsize global variable
about, 38
@@thresh_hysteresis global variable
about, 38
@@timeticks global variable
about, 38
@@total_errors global variable
about, 38
@@total_read global variable
about, 38
@@total_write global variable
about, 38
@@tranchained global variable
about, 38
@@trancount global variable
about, 38
@@transtate global variable
about, 38
@@version global variable
about, 38
@HttpMethod
HTTP header, 180
@HttpURI
HTTP header, 180
@HttpVersion

HTTP header, 180
@mp:id metaproperty
openxml system procedure, 835
@mp:localname metaproperty
openxml system procedure, 835
@mp:namespaceuri metaproperty
openxml system procedure, 835
@mp:prefix metaproperty
openxml system procedure, 835
@mp:xmltext metaproperty
openxml system procedure, 835
[ESQL]
statement indicators, 298
[Interactive SQL]
statement indicators, 298
[SP]
statement indicators, 298
[T-SQL]
statement indicators, 298
^
bitwise operator, 13
|
bitwise operator, 13
~
bitwise operator, 13
– comment indicator
about, 42

A

ABS function
SQL syntax, 103
ACCENT clause
CREATE DATABASE statement, 375
AccentSensitivity property
DB_EXTENDED_PROPERTY function, 143
AcceptCharset option
sa_set_http_option system procedure, 922
ACOS function
SQL syntax, 103
actions
referential integrity, 459
Adaptive Server Enterprise
converting stored procedures, 278
CREATE DATABASE statement, 375
migrating to SQL Anywhere using sa_migrate system procedure, 889
ADD PCTFREE clause
ALTER MATERIALIZED VIEW statement, 313
adding
columns using the ALTER TABLE statement, 332
indexes using the CREATE INDEX statement, 405
Java classes, 578
messages, 413
servers, 435
web services, 438
ADDRESS clause
CREATE SYNCHRONIZATION USER, 448
addresses
SQL Remote publishers, 319
AES encryption algorithm
CREATE DATABASE statement, 378
AFTER triggers
CREATE TRIGGER statement , 462
aggregate functions
alphabetical list, 93
aliases
for columns, 650
in the DELETE statement, 485
ALL
keyword in SELECT statement, 649
ALL search condition
SQL syntax, 21
ALLOCATE DESCRIPTOR statement
SQL syntax, 299
allocating
disk space using the ALTER DBSPACE statement, 305
memory for descriptor areas, 299
alphabetic characters
defined, 7
ALTER DATABASE statement
FORCE START clause, 303
SET PARTNER FAILOVER clause, 302
SQL syntax, 301
ALTER DATATYPE statement
SQL syntax, 307
ALTER DBSPACE statement
SQL syntax, 305
ALTER DOMAIN statement
SQL syntax, 307
ALTER EVENT statement
SQL syntax, 308
ALTER FUNCTION statement
SQL syntax, 310
ALTER INDEX statement

SQL syntax, 311
ALTER MATERIALIZED VIEW statement
SQL syntax, 313
ALTER PROCEDURE statement
SQL syntax, 315
ALTER PUBLICATION statement
SQL syntax, 317
ALTER REMOTE MESSAGE TYPE statement
SQL syntax, 319
ALTER SERVER statement
SQL syntax, 321
ALTER SERVICE statement
SQL syntax, 323
ALTER STATISTICS statement
SQL syntax, 327
ALTER SYNCHRONIZATION SUBSCRIPTION statement
SQL syntax, 328
ALTER SYNCHRONIZATION USER statement
SQL syntax, 330
ALTER TABLE statement
SQL syntax, 332
ALTER TRIGGER statement
SQL syntax, 341
ALTER VIEW statement
DISABLE clause, 342
ENABLE clause, 342
SQL syntax, 342
altering
ALTER PUBLICATION statement, 317
ALTER TABLE statement, 332
columns using the ALTER TABLE statement, 332
data types using the ALTER DOMAIN statement, 307
databases using the ALTER DATABASE statement, 301
dbspaces using the ALTER DBSPACE statement, 305
domains using the ALTER DOMAIN statement, 307
events using the ALTER EVENT statement, 308
functions using the ALTER FUNCTION statement, 310
indexes using the ALTER INDEX statement, 311
materialized views using the ALTER MATERIALIZED VIEW statement, 313
procedures using the ALTER PROCEDURE statement, 315
remote server attributes using the ALTER SERVER statement, 321
SQL Remote remote message types, 319
triggers using the ALTER TRIGGER statement, 341
views using the ALTER VIEW statement, 342
web services using the ALTER SERVICE statement, 323
ambiguous string to date conversions
about, 86
AND
bitwise operators, 13
logical operators description, 12
three-valued logic, 27
ANSI
equivalency using the REWRITE function, 236
ansi_nulls option
Microsoft SQL Server compatibility, 659
ansi_permissions
setting with Transact-SQL SET statement, 658
ansinull option
setting with Transact-SQL SET statement, 658
ANY search condition
SQL syntax, 21
apostrophes
in SQL strings, 9
application profiling
setting the tracing level, 925
approximate data types
about, 56
arc-cosine function
ACOS function, 103
arc-sine function
ASIN function, 105
arc-tangent function
ATAN function, 106
ATAN2 function, 106
archive backups
supported operating systems using the BACKUP statement, 346
archives
creating database backups using the BACKUP statement, 346
restoring databases from, 631
ARGN function
SQL syntax, 104
arithmetic
operators and SQL syntax, 12

arithmetic operators
Modulo, 13
articles
SYSARTICLE system view, 754
SYSARTICLECOL system view, 755
ASCII
function and SQL syntax, 104
ASE COMPATIBLE clause
CREATE DATABASE statement, 375
ASIN function
SQL syntax, 105
assigning
logins for remote servers, 397
values to SQL variables, 656
AT clause
create existing table, 396
ATAN function
SQL syntax, 106
ATAN2 function
SQL syntax, 106
ATN2 function
SQL syntax, 106
ATTACH TRACING statement
diagnostic tracing, 344
SQL syntax, 344
attributes
altering remote server using the ALTER SERVER statement, 321
auditing
adding comments, 841
auto_commit option
Interactive SQL option, 667
AUTOINCREMENT
@@identity, 41
CREATE TABLE statement, 454
GET_IDENTITY function, 168
autoincrement
resetting the value, 909
average function
AVG function, 107
AVG function
SQL syntax, 107

B

backslashes
in SQL strings, 9
not allowed in SQL identifiers, 7
BACKUP statement
SQL syntax, 346
backup.syb file
about, 347
backups
BACKUP permission, 548
BACKUP statement, 346
creating events using the CREATE EVENT statement, 390
creating using the BACKUP statement, 346
restoring databases from, 631
to tape using the BACKUP statement, 346
base 10 logarithm
LOG10 function, 196
base tables
CREATE TABLE statement, 459
BASE64_DECODE function
SQL syntax, 108
BASE64_ENCODE function
SQL syntax, 108
BEFORE triggers
CREATE TRIGGER statement , 462
BEGIN DECLARE statement
SQL syntax, 476
BEGIN keyword
compatibility, 352
BEGIN statement
SQL syntax, 351
BEGIN TRANSACTION statement
Transact-SQL syntax, 354
beginning
user-defined transactions using the BEGIN TRANSACTION statement, 354
BETWEEN clause
WINDOW clause, 720
BETWEEN search condition
SQL syntax, 22
BIGINT data type
syntax, 56
binary
escape characters, 9
binary constants (see binary literals)
BINARY data type
syntax, 74
BINARY data types
BINARY, 74
decoding, 108
encoding, 108

getting from columns, 542
IMAGE, 74
LONG BINARY, 75
UNIQUEIDENTIFIER, 75
VARBINARY, 76
binary data types
about, 74
binary large objects
BINARY data types, 74
exporting, 949
GET DATA SQL statement, 542
getting from columns, 542
importing ASE generated BCP files, 587
inserting using the xp_read_file system procedure, 947
SET statement example, 657
transaction log considerations, 305
binary literals
special characters, 9
BINARY VARYING data type (see VARBINARY data type)
bind variables
describing cursors, 490
EXECUTE SQL statement, 515
OPEN statement, 601
bit array conversions
about, 85
bit array data types
about, 65
LONG VARBIT, 65
VARBIT, 65
bit arrays
about, 65
converting, 85
data types, 65
BIT data type
syntax, 57
bit functions
alphabetical list, 94
BIT VARYING data type (see VARBIT data type)
BIT_AND function
SQL syntax, 110
BIT_LENGTH function
SQL syntax, 109
BIT_OR function
SQL syntax, 111
BIT_SUBSTR function
SQL syntax, 109
BIT_XOR function
SQL syntax, 112
bits
converting, 85
bitwise operators
SQL syntax, 13
blank padding
CREATE DATABASE statement, 375
BLOBs
BINARY data types, 74
exporting, 949
GET DATA SQL statement, 542
importing ASE generated BCP files, 587
indexing in CREATE TABLE statement, 453
INLINE clause, CREATE TABLE statement, 452
inserting using the SET statement, 656
inserting using the xp_read_file system procedure, 947
PREFIX clause, CREATE TABLE statement, 452
SET statement example, 657
transaction log considerations, 305
block fetches
FETCH statement, 527
OPEN statement, 602
blocking
identifying, 850
blocks
identifying, 850
troubleshooting, 882
brackets
database objects, 7
SQL identifiers, 7
BREAK statement
SQL syntax, 356
Transact-SQL syntax, 718
bugs
providing feedback, xix
bulk loading
LOAD TABLE statement, 585
bulk operations
unload, 698
unloading materialized views, 700
unloading tables, 700
BYE statement
SQL syntax, 522
bypassing optimization
avoiding using FORCE OPTIMIZATION option, 486, 513, 574, 580, 653, 695, 705

BYTE_LENGTH function
SQL syntax, 113
BYTE_SUBSTR function
SQL syntax, 114

C

cache
flushing, 868
CacheSizingStatistics property
setting with sa_server_option, 916
calibrating
cost models using the ALTER DATABASE statement, 301
parallel I/O capabilities, 303
the database server using the ALTER DATABASE statement, 301
CALL statement
in Transact-SQL, 517
SQL syntax, 357
calling procedures
CALL statement, 357
capabilities
remote servers, 756
SYSCAPABILITY system view, 755
CASE clause
CREATE DATABASE statement, 375
CASE expression
NULLIF function, 210
SQL syntax, 17
case sensitivity
comparison operators, 11
LIKE search condition, 23
CASE statement
SQL syntax, 359
CaseSensitivity property
DB_EXTENDED_PROPERTY function, 143
CAST function
data type conversions, 80
SQL syntax, 115
catalog
default system views, 753
system tables, 726
catalog procedures
alphabetical list , 835
catalog procedures (ASE)
sp_column_privileges, 963
sp_columns, 963
sp_fkeys, 963
sp_pkeys, 963
sp_special_columns, 963
sp_sproc_columns, 963
sp_statistics, 963
sp_stored_procedures, 963
sp_tables, 963
Transact-SQL list, 962
Transact-SQL, list, 963
catalog system procedures
about, 833
CatalogCollation property
DB_EXTENDED_PROPERTY function, 143
CEILING function
SQL syntax, 115
CHAR data type
byte-length semantics, 48
character-length semantics, 48
comparing with NCHAR data type, 81
syntax, 48
using DESCRIBE on a CHAR column, 48
CHAR function
SQL syntax, 116
CHAR VARYING data type (see VARCHAR data type)
CHAR_LENGTH function
SQL syntax, 118
character data
storage, 48
strings, 8
CHARACTER data type (see CHAR data type)
character data types
about, 48
CHAR, 48
LONG NVARCHAR, 49
LONG VARCHAR, 50
NCHAR, 50
NTEXT, 51
NVARCHAR, 52
TEXT, 53
UNIQUEIDENTIFIERSTR, 53
VARCHAR, 53
XML, 54
character functions
alphabetical list, 99
character set conversion
passwords, 549
when evaluating expressions, 81

character sets
COMPARE function, 119
converting during expression evaluation, 81
SORTKEY function, 249
character strings
about, 8
character substitution
about, 81
comparisons between CHAR and NCHAR, 81
different from character set to character set, 81
CHARACTER VARYING data type (see VARCHAR data type)
character-length semantics
CHAR data type, 48
VARCHAR data type, 53
CHARINDEX function
SQL syntax, 117
CharSet property
DB_EXTENDED_PROPERTY function, 143
CharsetConversion option
sa_set_http_option system procedure, 922
CHECK clause
search conditions, 20
CHECK conditions
CREATE TABLE statement, 456
CHECK CONSTRAINTS option
LOAD TABLE statement, 586
checkpoint logs
CHECKPOINT SQL statement, 361
CHECKPOINT statement
SQL syntax, 361
checkpointing
databases using the CHECKPOINT statement, 361
CHECKSUM clause
CREATE DATABASE statement, 375
checksums
creating databases with, 375
VALIDATE CHECKSUM statement, 713
validating, 713
classes
Java methods, 96
removing Java, 627
CLEAR statement
SQL syntax, 362
clearing
Interactive SQL panes, 362
CLOSE statement
SQL syntax, 363
close_on_endtrans option
setting with Transact-SQL SET statement, 658
closing
connections using the DROP CONNECTION statement, 500
cursors using the CLOSE statement [ESQL] [SP], 363
Interactive SQL, 522
clustered indexes
creating using ALTER INDEX statement, 311
COALESCE function
SQL syntax, 118
code pages
INPUT statement, 570
OUTPUT statement, 606
coefficient of determination
about, 227
COLLATION clause
collation tailoring, 376
CREATE DATABASE statement, 376
Collation property
DB_EXTENDED_PROPERTY function, 143
collation sequences, xi
(see also collations)
CREATE DATABASE statement, 376
LIKE search condition, 23
collation tailoring
COLLATION clause, CREATE DATABASE statement, 376
COMPARE function, 119
NCHAR COLLATION clause, CREATE DATABASE statement, 379
options, 376
SORTKEY function, 249
collations
SORTKEY function, 249
tailoring at database creation time, 376
CollectStatistics property
setting with sa_server_option, 916
column compression
ALTER TABLE statement, 332
CREATE TABLE statement, 450
retrieving compression statistics, 845
column constraints
adding using the ALTER TABLE statement, 334
changing using the ALTER TABLE statement, 336
column definition
CREATE TABLE statement, 452

column names
SQL syntax, 16
column names in expressions
about, 16
column statistics
only partially updated by LOAD TABLE, 590
selectivity estimates, 28
SYSCOLSTAT system view, 758
SYSCOLSTATS consolidated view, 811
updating with CREATE STATISTICS, 442
column-name
common element in SQL syntax, 295
columns
aliases, 650
altering using the ALTER TABLE statement, 332
constraints and defaults with domains, 78
constraints in CREATE TABLE statement, 456
domains, 78
getting binary data from, 542
permissions on, 757
renaming, 337
SYSTABCOL view, 797
updating, 710
updating without logging, 722
user-defined data types, 78
combining
result of multiple select statements, 695
comma-separated lists
LIST function syntax, 192
command files
parameters for Interactive SQL, 608
reading SQL statements from, 618
commands
executing operating system, 691
COMMENT statement
SQL syntax, 365
comments
adding to database objects using the COMMENT statement, 365
adding to services using the COMMENT statement, 365
SQL syntax, 42
COMMENTS INTRODUCED BY option
LOAD TABLE statement, 586
commit
preparing for two-phase, 612
COMMIT statement
referential integrity, 841
SQL syntax, 367
committing
transactions using the COMMIT statement, 367
common elements in SQL syntax
syntax, 295
communication protocols
multiple settings in MobiLink, 448
COMPARE function
collation tailoring, 119
SQL syntax, 119
comparing
CHAR and NCHAR, 81
COMPARE function, 119
comparing dates and times
about, 69
comparison operators
data conversion, 80
SQL syntax, 11
Transact-SQL compatibility, 11
comparisons
search conditions, 20
comparisons between CHAR and NCHAR
about, 81
compatibility
datetime, 68
NULLs, 44
T-SQL expressions and QUOTED IDENTIFIER option, 19
Transact-SQL comparison operators, 11
Transact-SQL expressions, 19
Transact-SQL global variables, 38
Transact-SQL local variables, 36
Transact-SQL views, 831
views, 824
compatibility of expressions
about, 19
compatibility views
about, 824
SYSCOLLATION, 824
SYSCOLLATIONMAPPINGS, 824
SYSCOLUMN, 825
SYSFKCOL, 825
SYSFOREIGNKEY, 826
SYSINDEX, 826
SYSINFO, 827
SYSIXCOL, 828
SYSTABLE, 828
SYSUSERLIST, 830

SYSUSERPERM, 830
SYSUSERPERMS, 831
compound statements
about, 351
compatibility, 352
COMPRESS function
SQL syntax, 121
compressed columns
ALTER TABLE statement, 332
retrieving compression statistics, 845
compressing
tables using the ALTER TABLE statement, 332
compressing columns
CREATE TABLE statement, 452
compressing strings on Unix
COMPRESS function, 121
compression
COMPRESS function, 121
statistics, 848
concatenating strings
string operators, 13
concurrency
locking tables, 593
condition
common element in SQL syntax, 295
conditions
EXISTS, 25
search, 20
SQL search conditions, 20
subqueries in, 21
three-valued logic, 27
CONFIGURE statement
SQL syntax, 369
CONFLICT function
SQL syntax, 123
conflicts
CONFLICT function for SQL Remote, 123
CONNECT statement
SQL syntax, 370
connecting
creating events using the CREATE EVENT statement, 390
databases using the COMMENT statement [ESQL] [Interactive SQL], 370
connection-level variables
definition, 36
SQL syntax, 37
connection-name
common element in SQL syntax, 295
CONNECTION_EXTENDED_PROPERTY function
SQL syntax, 121
CONNECTION_PROPERTY function
SQL syntax, 123
connections
creating events for failed connections, 390
disabling connections to a database, 914
disallowing with RAISERROR, 617
DROP CONNECTION statement, 500
dropping in Interactive SQL, 497
enabling pooling, 671
setting, 661
setting a maximum number, 617
ConnsDisabled property
setting with sa_server_option, 916
ConnsDisabledForDB property
setting with sa_server_option, 916
console
displaying messages on, 597
ConsoleLogFile property
setting with sa_server_option, 916
ConsoleLogMaxSize property
setting with sa_server_option, 916
CONSOLIDATE permissions
granting, 553
revoking, 638
consolidated databases
revoking permissions, 638
consolidated views
about, 809
SYSARTICLECOLS, 809
SYSARTICLES, 809
SYSCAPABILITIES, 810
SYSCATALOG, 810
SYSCOLAUTH, 810
SYSCOLSTATS, 811
SYSCOLUMNS, 811
SYSFOREIGNKEYS, 812
SYSGROUPS, 812
SYSINDEXES, 813
SYSOPTIONS, 814
SYSPROCAUTH, 814
SYSPROCPARMS, 815
SYSPROCS, 814
SYSPUBLICATIONS, 815
SYSREMOTEOPTION2, 816
SYSREMOTEOPTIONS, 816

SYSREMOTETYPES, 817
SYSREMOTEUSERS, 817
SYSSUBSCRIPTIONS, 818
SYSSYNC2, 818
SYSSYNCPUBLICATIONDEFAULTS, 819
SYSSYNCS, 819
SYSSYNCSCRIPTS, 819
SYSSYNCSUBSCRIPTIONS, 820
SYSSYNCUSERS, 821
SYSTABAUTH, 821
SYSTRIGGERS, 822
SYSUSERAUTH, 829
SYSUSEROPTIONS, 822
SYSVIEWS, 823
constant binary (see binary literals)
constant strings (see string literals)
constants (see binary literals) (see string literals)
about, 9
SQL syntax, 16
Transact-SQL, 19
constants in expressions
about, 16
constraints
column, CREATE TABLE statement, 456
renaming, 337
CONTINUE statement
SQL syntax, 373
Transact-SQL syntax, 718
control statements
BREAK syntax, 356
CALL SQL statement, 357
CASE SQL statement, 359
CONTINUE statement syntax, 373
GOTO Transact-SQL statement, 547
IF SQL statement, 563
LEAVE SQL statement, 582
LOOP SQL statement, 595
Transact-SQL BREAK statement, 718
Transact-SQL CONTINUE statement, 718
Transact-SQL IF statement, 565
Transact-SQL WHILE statement, 718
WHILE SQL statement, 595
conventions
documentation, xiv
file names in documentation, xvi
SQL language syntax, 4
syntax, 297
conversion
converting DOUBLE to NUMERIC, 86
data type conversions, 80
NCHAR to CHAR, 84
strings to dates, 68
when evaluating expressions, 80
conversion between character sets
about, 81
conversion functions
alphabetical list, 94
data type, 94
conversion when using comparison operators
about, 80
CONVERT function
data type conversions, 80
SQL syntax, 125
converting
ambiguous dates and strings, 86
bit arrays, 85
bits, 85
data types, 80
date to string, 84
SQL and Java, 88
using comparison operators, 80
converting between numeric sets
about, 86
converting dates to strings
about, 84
converting NULL constants to NUMERIC and string types
about, 84
converting strings
about, 99
coordinated universal time
UTC TIMESTAMP, 35
coordinated universal timestamp
CURRENT UTC TIMESTAMP, 32
copyright
retrieving, 960
CORR function
SQL syntax, 127
correlation function
CORR function, 127
correlation names
in the DELETE statement, 485
COS function
SQL syntax, 128
cosine function
COS function, 128

cost models
calibrating the database server, 301
loading, 881
recalibrating using the ALTER DATABASE statement, 301
unloading, 933
cost-based optimization
forcing for procedures, 486, 513, 575, 580, 653, 695, 705
forcing using FORCE OPTIMIZATION option, 486, 513, 574, 580, 653, 695, 705
COT function
SQL syntax, 129
cotangent function
COT function, 129
COUNT function
SQL syntax, 129
COUNT_SET_BITS function
SQL syntax, 130
COVAR_POP function
SQL syntax, 131
COVAR_SAMP function
SQL syntax, 132
CREATE DATABASE statement
SQL syntax, 374
CREATE DATATYPE statement
SQL syntax, 386
CREATE DBSPACE statement
SQL syntax, 382
CREATE DECRYPTED FILE statement
SQL syntax, 384
CREATE DOMAIN statement
SQL syntax, 386
using, 78
CREATE ENCRYPTED FILE statement
SQL syntax, 388
CREATE EVENT statement
SQL syntax, 390
CREATE EXISTING TABLE statement
sp_remote_columns system procedure, 938
sp_remote_tables system procedure, 944
SQL syntax, 395
CREATE EXTERNLOGIN statement
SQL syntax, 397
CREATE FUNCTION statement
SQL syntax, 399
Transact-SQL example, 404
CREATE INDEX statement
SQL syntax, 405
table use, 407
CREATE LOCAL TEMPORARY TABLE statement
SQL syntax, 409
CREATE MATERIALIZED VIEW statement
SQL syntax, 411
CREATE MESSAGE statement
Transact-SQL syntax, 413
CREATE PROCEDURE statement
SQL syntax, 414
Transact-SQL syntax, 425
CREATE PUBLICATION statement
SQL syntax, 427
CREATE REMOTE MESSAGE TYPE statement
SQL syntax, 431
CREATE SCHEMA statement
SQL syntax, 433
CREATE SERVER statement
SQL syntax, 435
CREATE SERVICE statement
SQL syntax, 438
CREATE STATISTICS statement
SQL syntax, 442
CREATE SUBSCRIPTION statement
SQL syntax, 443
CREATE SYNCHRONIZATION SUBSCRIPTION statement
SQL syntax, 445
CREATE SYNCHRONIZATION USER statement
SQL syntax, 448
CREATE TABLE statement
remote tables, 452
SQL syntax, 450
Transact-SQL, 460
CREATE TEMPORARY PROCEDURE statement
SQL syntax, 414
CREATE TRIGGER statement
SQL syntax, 462
Transact-SQL syntax, 468
CREATE VARIABLE statement
SQL syntax, 469
CREATE VIEW statement
SQL syntax, 471
creating
backups of databases using the BACKUP statement, 346
CREATE INDEX statement, 405
CREATE MATERIALIZED VIEW statement, 411

CREATE PUBLICATION statement, 427
CREATE SYNCHRONIZATION SUBSCRIPTION statement, 445
CREATE TABLE statement, 450
CREATE TRIGGER statement, 462
CREATE VIEW statement, 471
cursors, 478
cursors in Transact-SQL, 482
database files using the CREATE DBSPACE statement, 382
databases using the CREATE DATABASE statement, 374
functions using the CREATE FUNCTION statement, 399
local temporary tables, 483
local temporary tables using the CREATE LOCAL TEMPORARY TABLE statement, 409
messages, 413
proxy tables, 452
proxy tables using sp_remote_tables system procedure, 944
proxy tables using the CREATE EXISTING TABLE statement, 395
savepoints, 647
schemas, 433
servers, 435
SQL Remote remote message types, 431
SQL variables, 469, 477
stored procedures, 414
stored procedures in Transact SQL, 425
subscriptions, 443
triggers in Transact-SQL, 468
web services, 438
creating domains
CREATE DOMAIN statement, 386
CROSS JOIN clause
FROM clause SQL syntax, 535
CSCONVERT function
SQL syntax, 133
CUBE operation
GROUP BY clause, 560
WITH CUBE clause, 560
CUME_DIST function
SQL syntax, 135
CURRENT DATABASE
special value, 30
CURRENT DATE
special value, 30
current date function
TODAY function, 268
CURRENT PUBLISHER
setting, 555
special value, 30
CURRENT TIME
special value, 31
CURRENT TIMESTAMP
special value, 31
CURRENT USER
special value, 32
CURRENT UTC TIMESTAMP
special value, 32
CURRENT_TIMESTAMP
special value, 31
CURRENT_USER
special value, 32
cursors
CLOSE statement [ESQL] [SP], 363
declaring, 478
declaring in Transact-SQL, 482
deleting rows from, 488
describing, 490
describing behavior, 417
EXPLAIN statement syntax, 524
fetching rows from, 526
inserting rows using, 614
looping over, 530
opening, 601
preparing statements, 610
re-describing, 417
updatability set in SELECT statement, 652

D

data
exporting from tables into files, 604
importing into tables from files, 568
selecting rows, 648
data access plans
getting text specification, 524
data type conversion
comparing CHAR and NCHAR values, 81
comparison operators, 80
converting DOUBLE to NUMERIC, 86
converting NCHAR to CHAR, 84
Java-to-SQL, 88
SQL to Java, 88

SQL-to-Java, 89
when evaluating expressions, 80
data type conversion functions
about, 94
data type conversions
about, 80
data types
about, 47
altering using the ALTER DOMAIN statement, 307
BIGINT, 56
BINARY, 74
BIT, 57
bit array, 65
BIT VARYING (VARBIT), 65
CHAR, 48
CHAR VARYING (VARCHAR), 53
character, 48
CHARACTER (CHAR), 48
CHARACTER VARYING (VARCHAR), 53
comparing values, 80
compatibility, 68
converting for comparison operators, 80
converting Java and SQL, 88
CREATE DOMAIN statement, 386
date, 67
DATE, 71
DATETIME, 72
DEC (DECIMAL), 57
DECIMAL, 57
DOUBLE, 58
dropping user-defined using the DROP statement, 498
FLOAT, 59
IMAGE, 74
INT (INTEGER), 60
INTEGER, 60
LONG BINARY, 75
LONG BIT VARYING (LONG VARBIT), 65
LONG NVARCHAR, 49
LONG VARBIT, 65
LONG VARCHAR, 50
MONEY, 64
NATIONAL CHAR (NCHAR), 50
NATIONAL CHAR VARYING (NVARCHAR), 52
NATIONAL CHARACTER (NCHAR), 50
NATIONAL CHARACTER VARYING (NVARCHAR), 52
NCHAR, 50
NCHAR VARYING (NVARCHAR), 52
NTEXT, 51
numeric, 56
NUMERIC, 61
NVARCHAR, 52
REAL, 62
retrieving, 164
roundoff errors, 56
SMALLDATETIME, 72
SMALLINT, 62
SMALLMONEY, 64
SQL conversion functions, 94
SYSDOMAIN system view, 761
SYSEXTERNLOGIN system view, 763
SYSUSERTYPE system view, 805
TEXT, 53
time, 67
TIME, 72
TIMESTAMP, 73
TINYINT, 63
Unicode, 48
UNIQUEIDENTIFIER, 75
UNIQUEIDENTIFIERSTR, 53
user-defined domains, 78
VARBINARY, 76
VARBIT, 65
VARCHAR, 53
XML, 54
data-type
common element in SQL syntax, 295
database cleaner
about, 843
sa_clean_database system procedure, 842
database extraction
REMOTE RESET statement (SQL Remote), 626
database files
dropping using the DROP DATABASE statement, 501
storing indexes in, 406
database ID numbers
DB_ID function, 146
database mirroring
initiating a failover, 302
database names
DB_NAME function, 147

database objects
adding comments using the COMMENT statement, 365
identifying, 7
database options
date_order and unambiguous dates, 70
initial settings and sp_login_environment system procedure, 936
initial settings and sp_tsql_environment system procedure, 945
quoted_identifier and T-SQL compatibility, 19
setting in Transact-SQL, 658
Transact-SQL compatibility, 945
database schemas
system tables, 726
system views, 753
database servers
setting options with sa_server_option system procedure, 914
START ENGINE statement, 676
STOP ENGINE statement, 684
DATABASE SIZE clause
CREATE DATABASE statement, 378
database validation
VALIDATE CHECKSUM statement, 713
VALIDATE INDEX statement, 713
DatabaseCleaner property
setting with sa_server_option, 917
databases
backing up using the BACKUP statement, 346
checkpointing using the CHECKPOINT statement, 361
connecting to using the COMMENT statement [ESQL] [Interactive SQL], 370
creating files using the CREATE DBSPACE statement, 382
creating using the CREATE DATABASE statement, 374
default system views, 753
disabling connections, 914
dropping files using the DROP DATABASE statement, 501
loading bulk data into, 585
migrating, 888
restoring from archives, 631
schema, 726, 753
starting, 674
stopping, 683
structure, 726, 753
SYSFILE system view, 764
system procedures, 833
system tables, 726
unloading data, 698
unloading materialized views, 700
unloading tables, 700
upgrading jConnect using the ALTER DATABASE statement, 301
validating, 934
DATALENGTH function
SQL syntax, 136
DATATYPE clause
ALTER SERVICE statement, 324
CREATE SERVICE statement, 439
date and time data types
about, 67
TIME, 72
TIMESTAMP, 73
DATE data type
syntax, 71
date data types
about, 67
DATE, 71
DATETIME, 72
SMALLDATETIME, 72
DATE function
SQL syntax, 136
date functions
alphabetical list, 94
date parts
about, 95
date_order option
ODBC, 70
using, 70
DATEADD function
SQL syntax, 137
DATEDIFF function
SQL syntax, 137
datefirst option
SET statement syntax, 658
DATEFORMAT function
SQL syntax, 139
DATENAME function
SQL syntax, 139
DATEPART function
SQL syntax, 140
dates

ambiguous string conversions, 84, 86
comparing, 69
conversion functions, 94
conversion problems, 84
converting from strings, 68
February 29, 69
generating table of, 912
inserting, 71
interpretation, 71
interpreting strings as dates, 70
leap years, 69
queries, 68
retrieving, 71
sending to the database, 67
SQL Anywhere, 67
storing, 67
unambiguous specification of, 70
datetime
conversion functions, 94
DATETIME data type
syntax, 72
DATETIME function
SQL syntax, 141
DAY function
SQL syntax, 141
day of week
DOW function, 153
DAYNAME function
SQL syntax, 141
DAYS function
SQL syntax, 142
DB2
migrating to SQL Anywhere using sa_migrate system procedure, 889
DB_EXTENDED_PROPERTY function
SQL syntax, 143
DB_ID function
SQL syntax, 146
DB_NAME function
SQL syntax, 147
DB_PROPERTY function
SQL syntax, 147
db_register_a_callback function
using with MESSAGE TO CLIENT, 599
DBA PASSWORD clause
CREATE DATABASE statement, 378
DBA USER clause
CREATE DATABASE statement, 378
DBFreePercent event condition
about, 158
DBFreeSpace event condition
about, 158
dbo user
RowGenerator system table, 751
Transact-SQL compatibility views, 831
DBSize event condition
about, 158
dbspaces
altering using the ALTER DBSPACE statement, 305
creating using the CREATE DBSPACE statement, 382
determining available space, 865
dropping using the DROP statement, 498
SYSFILE system view, 764
deadlock reporting
sa_report_deadlocks system procedure, 908
deadlocks
sa_report_deadlocks system procedure, 908
DEALLOCATE DESCRIPTOR statement
SQL syntax, 475
DEALLOCATE statement
SQL syntax, 474
deallocating
descriptor areas, 475
debugging
controlling MESSAGE statement behavior, 597
DebuggingInformation property
setting with sa_server_option, 917
DEC data type (see DECIMAL data type)
DECIMAL data type
syntax, 57
DECLARE CURSOR statement
SQL syntax, 478
Transact-SQL syntax, 482
DECLARE EXCEPTION
used with BEGIN statement, 351
DECLARE LOCAL TEMPORARY TABLE statement
SQL syntax, 483
DECLARE statement
SQL syntax, 477
used with BEGIN statement, 351
declaring
cursors, 478
cursors in Transact-SQL, 482
host variables in embedded SQL, 476

variables SQL, 477
decoding data
BASE64_DECODE function, 108
HTML_DECODE function, 176
DECOMPRESS function
SQL syntax, 148
decompressing strings on Unix
DECOMPRESS function, 148
decompression
DECOMPRESS function, 148
DECRYPT function
SQL syntax, 149
decrypting
files using the CREATE DECRYPTED FILE statement, 384
tables using the ALTER TABLE statement, 332
DEFAULT LAST USER
avoid replicating columns in SQL Remote, 574
DEFAULT TIMESTAMP columns
about, 454
TIMESTAMP special value, 33
default values
CURRENT DATABASE, 30
CURRENT DATE, 30
CURRENT PUBLISHER, 30
CURRENT TIME, 31
CURRENT TIMESTAMP, 31
CURRENT USER, 32
CURRENT UTC TIMESTAMP, 32
CURRENT_TIMESTAMP, 31
CURRENT_USER, 32
LAST USER, 32
SQLCODE, 33
SQLSTATE, 33
TIMESTAMP, 33
USER, 34
UTC TIMESTAMP, 35
defaults
CREATE TABLE statement, 454
DEFAULTS option
LOAD TABLE statement, 586
definitions
altering tables using the ALTER TABLE statement, 332
defragmenting
REORGANIZE TABLE, 628
DEGREES function
SQL syntax, 150
DELETE (positioned) statement
SQL syntax, 488
DELETE statement
SQL syntax, 485
deleting
all rows from a table, 693
columns using the ALTER TABLE statement, 332
database files using the DROP DATABASE statement, 501
dbspaces using the DROP statement, 498
domains, 498
events using the DROP statement, 498
functions using the DROP statement, 498
granting permissions, 636
indexes using the DROP statement, 498
Java classes, 627
materialized views using the DROP statement, 498
optimizer statistics using the DROP STATISTICS statement, 508
prepared statements using the DROP STATEMENT statement, 507
procedures using the DROP statement, 498
rows from cursors, 488
rows from databases, 485
SQL variables using the DROP VARIABLE statement, 512
START SYNCHRONIZATION DELETE statement, 681
STOP SYNCHRONIZATION DELETE statement, 688
tables using the DROP statement, 498
triggers using the DROP statement, 498
views using the DROP statement, 498
DELETING condition
triggers, 26
DELIMITED BY option
LOAD TABLE statement, 587
delimited strings
compatibility with ASE, 19
delimiting SQL strings
about, 7
DENSE_RANK function
SQL syntax, 151
denying
granting permissions, 636
dependencies
determining using sa_dependent_views system procedure, 859

dependent variables
regression line, 223
derived tables
FROM clause SQL syntax, 535
lateral, 537
DESCRIBE statement
Interactive SQL syntax, 494
long column names, 491
SQL syntax, 490
describing
cursor behavior, 417
cursors, 490
descriptor areas
allocating memory for, 299
deallocating, 475
EXECUTE SQL statement, 515
getting information from, 544
setting, 662
UPDATE (positioned) statement, 708
descriptors
DESCRIBE statement, 490
FETCH SQL statement, 526
preparing statements, 610
DETACH TRACING statement
diagnostic tracing, 496
SQL syntax, 496
deterministic functins
CREATE FUNCTION statement, 399
developer community
newsgroups, xix
diagnostic tracing
ATTACH TRACING statement, 344
DETACH TRACING statement, 496
REFRESH TRACING LEVEL statement, 623
sa_diagnostic_auxiliary_catalog table, 735
sa_diagnostic_blocking table, 736
sa_diagnostic_cachecontents table, 737
sa_diagnostic_connection table, 738
sa_diagnostic_cursor table, 739
sa_diagnostic_deadlock table, 740
sa_diagnostic_hostvariable table, 741
sa_diagnostic_internalvariable table, 742
sa_diagnostic_query table, 743
sa_diagnostic_request table, 745
sa_diagnostic_statement table, 747
sa_diagnostic_statistics table, 747
sa_diagnostic_tracing_level table, 748
tables, about, 735
diagnostic tracing level
setting at the command-line, 925
diagnostics
sa_performance_statistics system procedure, 901
DIFFERENCE function
SQL syntax, 152
directory access servers
CREATE SERVER statement, 435
DISABLE clause
ALTER MATERIALIZED VIEW statement, 313
ALTER VIEW statement, 342
DISABLE USE IN OPTIMIZATION clause
ALTER MATERIALIZED VIEW statement, 313
disabling connections
to all databases on a server, 916
to individual databases, 916
DISCONNECT statement
SQL syntax, 497
disconnecting
creating events using the CREATE EVENT statement, 390
DROP CONNECTION statement, 500
DISH services
forward slashes not allowed in name, 438
disk space
creating events using the CREATE EVENT statement, 390
creating out of disk space events, 390
disk transfer time model
calibrating using the ALTER DATABASE statement, 301
restoring the default using the ALTER DATABASE statement, 301
displaying
messages, 597
messages in the message window, 613
DISTINCT clause
NULL, 44
DISTINCT keyword
about, 649
documentation
conventions, xiv
conventions for SQL syntax, 295
SQL Anywhere, xii
domains
about, 78
altering using the ALTER DOMAIN statement, 307

CREATE DOMAIN statement, 386
dropping using the DROP statement, 498
nullability, 386
Transact-SQL, 79
DOUBLE data type
converting to NUMERIC, 86
syntax, 58
double hyphen
comment indicator, 42
double quotes
database objects, 7
not allowed in SQL identifiers, 7
double slash
comment indicator, 42
DOW function
SQL syntax, 153
download-only
CREATE PUBLICATION syntax, 427
DriveType property
DB_EXTENDED_PROPERTY function, 143
DROP CONNECTION statement
SQL syntax, 500
DROP DATABASE statement
SQL syntax, 501
DROP DATATYPE statement
SQL syntax, 498
DROP DBSPACE statement
SQL syntax, 498
DROP DOMAIN statement
SQL syntax, 498
DROP EVENT statement
SQL syntax, 498
DROP EXTERNLOGIN statement
SQL syntax, 502
DROP FUNCTION statement
SQL syntax, 498
DROP INDEX statement
SQL syntax, 498
DROP MATERIALIZED VIEW statement
SQL syntax, 498
DROP MESSAGE statement
SQL syntax, 498
DROP PCTFREE clause
ALTER MATERIALIZED VIEW statement, 313
DROP PROCEDURE statement
SQL syntax, 498
DROP PUBLICATION statement
SQL syntax, 503
DROP REMOTE MESSAGE TYPE statement
SQL syntax, 504
DROP SERVER statement
SQL syntax, 505
DROP SERVICE statement
SQL syntax, 506
DROP statement
SQL syntax, 498
DROP STATEMENT statement
SQL syntax, 507
DROP STATISTICS statement
SQL syntax, 508
DROP SUBSCRIPTION statement
SQL syntax, 509
DROP SYNCHRONIZATION SUBSCRIPTION statement
SQL syntax, 510
DROP SYNCHRONIZATION USER statement
SQL syntax, 511
DROP TABLE statement
SQL syntax, 498
DROP TRIGGER statement
SQL syntax, 498
DROP VARIABLE statement
SQL syntax, 512
DROP VIEW statement
SQL syntax, 498
dropping
columns using the ALTER TABLE statement, 332
connections in Interactive SQL, 497
connections using the DROP CONNECTION statement, 500
database files using the DROP DATABASE statement, 501
dbspaces using the DROP statement, 498
domains, 498
DROP PUBLICATION statement, 503
DROP SUBSCRIPTION statement, 509
DROP SYNCHRONIZATION SUBSCRIPTION statement, 510
DROP SYNCHRONIZATION USER statement, 511
events using the DROP statement, 498
functions using the DROP statement, 498
indexes using the DROP statement, 498
logins for remote servers, 502
materialized views using the DROP statement, 498

optimizer statistics using the DROP STATISTICS statement, 508
prepared statements using the DROP STATEMENT statement, 507
procedures using the DROP statement, 498
remote message types, 504
remote servers using the DROP SERVER statement, 505
SQL variables using the DROP VARIABLE statement, 512
tables using the DROP statement, 498
triggers using the DROP statement, 498
users, 636
views using the DROP statement, 498
web services using the DROP SERVICE statement, 506
DUMMY
system table, 726
DUMMY system table
Row Constructor algorithm, 726
DYNAMIC SCROLL cursors
declaring, 478
dynamic SQL
executing procedures in, 519

E

elements
SQL language syntax, 4
ELSE
CASE expression, 17
IF expressions, 17
email
extended system procedures, 951
system procedures, 953
embedded SQL
ALLOCATE DESCRIPTOR syntax, 299
BEGIN DECLARE statement syntax, 476
CLOSE statement syntax, 363
CONNECT statement syntax, 370
DEALLOCATE DESCRIPTOR statement syntax, 475
DECLARE CURSOR statement syntax, 478
DELETE (positioned) statement syntax, 488
DESCRIBE statement syntax, 490
DISCONNECT statement syntax, 497
DROP STATEMENT statement syntax, 507
END DECLARE statement syntax, 476
EXECUTE IMMEDIATE statement syntax, 519
EXECUTE statement syntax, 515
EXPLAIN statement syntax, 524
FETCH statement syntax, 526
GET DATA statement syntax, 542
GET DESCRIPTOR statement syntax, 544
GET OPTION statement syntax, 546
INCLUDE statement syntax, 567
OPEN statement syntax, 601
PREPARE statement syntax, 610
PUT statement syntax, 614
SET CONNECTION statement syntax, 661
SET DESCRIPTOR statement syntax, 662
SET SQLCA statement syntax, 670
WHENEVER statement syntax, 717
ENABLE clause
ALTER MATERIALIZED VIEW statement, 313
ALTER VIEW statement, 342
ENABLE USE IN OPTIMIZATION clause
ALTER MATERIALIZED VIEW statement, 313
encoding
INPUT statement, 570
LOAD TABLE syntax, 585
OUTPUT statement, 606
READ statement, 618
UNLOAD TABLE syntax, 700
ENCODING clause
CREATE DATABASE statement, 378
encoding data
BASE64_ENCODE function, 108
HTML_ENCODE function, 177
ENCODING option
LOAD TABLE statement, 587
UNLOAD TABLE statement, 700
encodings
CREATE DATABASE statement, 376
ENCRYPT function
SQL syntax, 154
ENCRYPTED clause
ALTER MATERIALIZED VIEW statement, 313
CREATE DATABASE statement, 378
ENCRYPTED TABLE clause
CREATE DATABASE statement, 378
encrypting tables
ALTER TABLE statement, 332
encryption
CREATE ENCRYPTED FILE statement, 388
database files, 378

encryption algorithms
CREATE DATABASE statement, 378
END
CASE expression, 17
END DECLARE statement
SQL syntax, 476
END keyword
compatibility, 352
END LOOP statement
SQL syntax, 595
END statement
used with BEGIN statement, 351
ENDIF
IF expressions, 17
ending
rolling back transactions, 642
engines
starting database, 676
stopping database, 684
error messages
ERRORMSG function, 155
ERRORMSG function
SQL syntax, 155
ErrorNumber event condition
about, 158
errors
creating events using the CREATE EVENT statement, 390
raising in Transact-SQL, 616
signaling, 673
trapping in embedded SQL, 717
user-defined messages, 804
escape character
INPUT SQL statement, 568
OUTPUT SQL statement, 604
ESCAPE CHARACTER option
LOAD TABLE statement, 587
escape characters
about , 9
binary literals, 9
escape sequences
backslashes in SQL strings, 9
hexadecimal values in SQL strings, 9
new line characters in SQL strings, 9
single quotes in SQL strings, 9
ESCAPES option
LOAD TABLE statement, 587
ESQL
statement indicators, 298
establishing
savepoints, 647
ESTIMATE function
SQL syntax, 156
ESTIMATE_SOURCE function
SQL syntax, 156
estimates
explicit selectivity estimates, 28
event conditions
list, 158
EVENT_CONDITION function
SQL syntax, 158
EVENT_CONDITION_NAME function
SQL syntax, 159
EVENT_PARAMETER function
SQL syntax, 160
events
altering using the ALTER EVENT statement, 308
creating and scheduling, 390
dropping using the DROP statement, 498
EVENT_PARAMETER, 160
scheduling using the ALTER EVENT statement, 308
scheduling using the CREATE EVENT statement, 390
triggering, 692
EXCEPT statement
SQL syntax, 513
EXCEPTION clause
BEGIN statement, 351
exceptions
resignaling, 630
signaling, 673
exclusive OR
bitwise operator, 13
EXECUTE IMMEDIATE statement
SQL syntax, 519
EXECUTE statement
SQL syntax, 515
Transact-SQL syntax, 517
executing
operating system commands, 691
prepared statements, 515
resuming execution of procedures, 633
SQL statements from files, 618
stored procedures in Transact-SQL, 517
EXISTS search condition

SQL syntax, 25
exit codes
EXIT statement [Interactive SQL], 522
EXIT statement
SQL syntax, 522
exiting
Interactive SQL, 522
procedures, 634
EXP function
SQL syntax, 162
EXPERIENCE_ESTIMATE function
SQL syntax, 162
EXPLAIN statement
SQL syntax, 524
EXPLANATION function
SQL syntax, 163
explicit selectivity estimates
about, 28
exponential function
EXP function, 162
exporting
BLOBs, 949
unloading materialized views, 700
unloading result sets, 698
unloading tables, 700
exporting data
tables into files, 604
expression
common element in SQL syntax, 295
expressions
CASE expressions, 17
column names, 16
constants, 16
data types of, 164
IF expressions, 17
SQL operator precedence, 14
subqueries, 16
syntax, 15
Transact-SQL compatibility, 19
EXPRTYPE function
SQL syntax, 164
extended procedures
about, 951
external function calls
procedures, 418
external functions
Java example, 404
external logins
assigning for remote servers, 397
dropping for remote servers, 502
EXTERNAL NAME clause
CREATE PROCEDURE statement, 418

F

FALSE conditions
IS FALSE search condition, 26
three-valued logic, 27
FASTFIRSTROW table hint
FROM clause, 539
February 29
about, 69
feedback
documentation, xix
providing, xix
FETCH statement
SQL syntax, 526
fetching
rows from cursors, 526
FILE message type
DROP REMOTE MESSAGE TYPE statement, 504
SQL Remote ALTER REMOTE MESSAGE TYPE statement, 319
SQL Remote CREATE REMOTE MESSAGE TYPE statement, 431
File property
DB_EXTENDED_PROPERTY function, 143
file size
creating events using the CREATE EVENT statement, 390
file-name
common element in SQL syntax, 295
files
allocating space for database, 305
creating database using the CREATE DBSPACE statement, 382
decrypting using the CREATE DECRYPTED FILE statement, 384
encrypting using the CREATE ENCRYPTED FILE statement, 388
exporting data from tables into, 604
importing data into tables from, 568
reading SQL statements from, 618
xp_read_file system procedure, 947
xp_write_file system procedure, 949

FileSize property
DB_EXTENDED_PROPERTY function, 143
finding out more and providing feed back
technical support, xix
FIRST clause
SELECT statement, 648
FIRST_VALUE function
SQL syntax, 165
FLOAT data type
syntax, 59
FLOOR function
SQL syntax, 167
FOLLOWING clause
WINDOW clause, 720
FOR clause
SELECT statement, 652
FOR OLAP WORKLOAD option
ALTER TABLE statement, 332
CREATE INDEX statement, 406
CREATE TABLE statement, 459
FOR statement
SQL syntax, 530
FOR UPDATE clause
SELECT statement syntax, 652
FOR UPLOAD clause
CREATE PUBLICATION statement, 427
FOR XML clause
SELECT statement, 648
FORCE INDEX
index hints, 536
FORCE OPTIMIZATION option
DELETE statement, 486
EXCEPT statement, 513
INSERT statement, 574
INTERSECT statement, 580, 695
SELECT statement, 653
UPDATE statement, 705
using in procedures, 486, 513, 575, 580, 653, 695, 705
FORCE START clause
ALTER DATABASE statement, 303
foreign keys
ALTER INDEX statement, 311
clustering using the ALTER INDEX statement, 311
consolidated views, 812
integrity constraints in CREATE TABLE statement, 456
remote tables, 938, 940
renaming using the ALTER INDEX statement, 311
system views, 765
unnamed in CREATE TABLE statement, 456
foreign tables
system views, 765
forest
defined, 284
FORMAT option
LOAD TABLE statement, 587
FORWARD TO statement
SQL syntax, 533
fragmentation
tables, 628
FreePages property
DB_EXTENDED_PROPERTY function, 143
frequency
sending messages, 553, 556
FROM clause
SELECT statement, 651
selecting from stored procedures, 536
SQL syntax, 535
FTP message type
SQL Remote ALTER REMOTE MESSAGE TYPE statement, 319
SQL Remote CREATE REMOTE MESSAGE TYPE statement, 431
functions
aggregate, 93
alphabetical list of all functions, 103
altering using the ALTER FUNCTION statement, 310
bit array, 94
creating using the CREATE FUNCTION statement, 399
data type conversion SQL, 94
date and time, 94
dropping using the DROP statement, 498
exiting from user-defined, 634
HTTP, 98
image SQL, 102
indexes on, 406
introduction, 91
Java, 96
miscellaneous, 97
numeric, 98
ranking, 94
returning values from user-defined, 634

SOAP, 98
string, 99
system, 100
text SQL, 102
types of functions, 93
user-defined, 96
functions, aggregate
about, 93
AVG, 107
BIT_AND, 110
BIT_OR, 111
BIT_XOR, 112
COUNT, 129
FIRST_VALUE, 165
GROUPING, 171
LAST_VALUE, 187
LIST, 192
MAX, 198
MIN, 199
SET_BITS, 245
STDDEV, 257
STDDEV_POP, 257
STDDEV_SAMP, 258
SUM, 264
VAR_POP, 275
VAR_SAMP, 276
VARIANCE, 278
functions, bit
GET_BIT, 167
functions, bit array
about, 94
BIT_LENGTH, 109
BIT_SUBSTR, 109
COUNT_SET_BITS, 130
SET_BIT, 244
functions, data type conversion
about, 94
CAST, 115
CONVERT, 125
HEXTOINT, 173
INTTOHEX, 184
ISDATE, 185
ISNULL, 186
functions, date and time
about, 94
DATE, 136
DATEADD, 137
DATEDIFF, 137
DATEFORMAT, 139
DATENAME, 139
DATEPART, 140
DATETIME, 141
DAY, 141
DAYNAME, 141
DAYS, 142
DOW, 153
GETDATE, 169
HOUR, 174
HOURS, 175
MINUTE, 199
MINUTES, 200
MONTH, 202
MONTHNAME, 202
MONTHS, 203
NOW, 210
QUARTER, 218
SECOND, 242
SECONDS, 243
TODAY, 268
WEEKS, 279
YEAR, 286
YEARS, 286
YMD, 288
functions, HTTP
about, 98
HTTP_HEADER, 179
HTTP_VARIABLE, 181
NEXT_HTTP_HEADER, 207
NEXT_HTTP_VARIABLE, 208
functions, Java and SQL user-defined
about, 96
functions, miscellaneous
about, 97
ARGN, 104
COALESCE, 118
CONFLICT, 123
ERRORMSG, 155
ESTIMATE, 156
ESTIMATE_SOURCE, 156
EXPERIENCE_ESTIMATE, 162
EXPLANATION, 163
GET_IDENTITY, 168
GRAPHICAL_PLAN, 169
GREATER, 171
IDENTITY, 182
IFNULL, 182

INDEX_ESTIMATE, 183
ISNUMERIC, 186
LESSER, 191
NEWID, 204
NULLIF, 210
NUMBER, 211
PLAN, 214
REWRITE, 236
SHORT_PLAN, 246
SQLDIALECT, 255
TRACEBACK, 268
TRACED_PLAN, 269
TRANSACTSQL, 269
VAREXISTS, 278
WATCOMSQL, 278
functions, numeric
about, 98
ABS, 103
ACOS, 103
ASIN, 105
ATAN, 106
ATAN2, 106
ATN2, 106
CEILING, 115
CONNECTION_PROPERTY, 123
COS, 128
COT, 129
DEGREES, 150
EXP, 162
FLOOR, 167
LOG, 195
LOG10, 196
MOD, 201
PI, 214
POWER, 215
RADIANS, 219
RAND, 219
REMAINDER, 233
ROUND, 239
SIGN, 246
SIN, 248
SQRT, 256
TAN, 265
TRUNCNUM, 270
functions, ranking
about, 94
functions, SOAP
about, 98
NEXT_SOAP_HEADER, 209
SOAP_HEADER, 248
functions, string
about, 99
ASCII, 104
BYTE_LENGTH, 113
BYTE_SUBSTR, 114
CHAR, 116
CHAR_LENGTH, 118
CHARINDEX, 117
COMPARE, 119
COMPRESS function, 121
CONNECTION_EXTENDED_PROPERTY, 121
CSCONVERT, 133
DECOMPRESS function, 148
DECRYPT function, 149
DIFFERENCE, 152
ENCRYPT function, 154
HASH function, 172
INSERTSTR, 184
LCASE, 189
LEFT, 190
LENGTH, 190
LOCATE, 194
LOWER, 196
LTRIM, 197
NCHAR, 204
PATINDEX, 212
REPEAT, 233
REPLACE, 234
REPLICATE, 235
REVERSE, 236
RIGHT, 238
RTRIM, 242
SIMILAR, 247
SORTKEY, 249
SOUNDEX, 253
SPACE, 254
STR, 259
STRING, 260
STRTOUUID, 261
STUFF, 262
SUBSTRING, 262
TO_CHAR, 266
TO_NCHAR, 267
TRIM, 270
UCASE, 271
UNICODE, 272

UNISTR, 272
UPPER, 273
UUIDTOSTR, 274
XMLAGG, 280
XMLCONCAT, 281
XMLELEMENT, 282
XMLFOREST, 284
XMLGEN, 285
functions, system
DATALENGTH, 136
DB_EXTENDED_PROPERTY, 143
DB_ID, 146
DB_NAME, 147
DB_PROPERTY, 147
EVENT_CONDITION, 158
EVENT_CONDITION_NAME, 159
EVENT_PARAMETER, 160
NEXT_CONNECTION, 205
NEXT_DATABASE, 207
PROPERTY, 216
PROPERTY_DESCRIPTION, 217
PROPERTY_NAME, 217
PROPERTY_NUMBER, 218
functions, text and image
about, 102
TEXTPTR, 265

G

GET DATA statement
SQL syntax, 542
GET DESCRIPTOR statement
SQL syntax, 544
GET OPTION statement
SQL syntax, 546
GET_BIT function
SQL syntax, 167
GET_IDENTITY function
SQL syntax, 168
GETDATE function
SQL syntax, 169
getting
binary data from columns, 542
information from descriptor areas, 544
option values, 546
getting help
technical support, xix
GLOBAL AUTOINCREMENT
CREATE TABLE statement, 454
global autoincrement
creating events using the CREATE EVENT statement, 390
global temporary tables
CREATE TABLE statement, 450
global variables
@@identity, 41
alphabetical list, 38
definition, 36
triggers and @@identity, 41
global_database_id option
CREATE TABLE statement, 454
globally unique identifiers
SQL syntax for NEWID function, 204
goodness of fit
regression lines, 227
GOTO statement
Transact-SQL syntax, 547
GRANT ALL statement
SQL syntax, 548
GRANT ALTER statement
SQL syntax, 548
GRANT BACKUP statement
SQL syntax, 548
GRANT CONNECT statement
SQL syntax, 548
GRANT CONSOLIDATE statement
SQL syntax, 553
GRANT DBA statement
SQL syntax, 548
GRANT DELETE statement
SQL syntax, 548
GRANT EXECUTE statement
SQL syntax, 548
GRANT GROUP statement
SQL syntax, 548
GRANT INSERT statement
SQL syntax, 548
GRANT INTEGRATED LOGIN statement
SQL syntax, 548
GRANT KERBEROS LOGIN statement
SQL syntax, 548
GRANT MEMBERSHIP IN GROUP statement
SQL syntax, 548
GRANT PUBLISH statement
SQL syntax, 555
GRANT REFERENCES statement

SQL syntax, 548
GRANT REMOTE DBA statement
SQL syntax, 558
GRANT REMOTE statement
SQL syntax, 556
GRANT RESOURCE statement
SQL syntax, 548
GRANT SELECT statement
SQL syntax, 548
GRANT statement
reviewing permissions, 757
SQL syntax, 548
GRANT UPDATE statement
SQL syntax, 548
GRANT VALIDATE statement
SQL syntax, 548
granting
CONSOLIDATE permissions, 553
permissions, 548
PUBLISH permissions, 555
remote DBA permissions, 558
REMOTE permissions, 556
GRAPHICAL_PLAN function
SQL syntax, 169
GREATER function
SQL syntax, 171
GROUP BY clause
CUBE operation, 560
GROUPING SETS operation, 559
ROLLUP operation, 560
SELECT statement, 651
SQL syntax, 559
grouping
statements in a BEGIN statement, 351
GROUPING function
SQL syntax, 171
GROUPING SETS operation
GROUP BY clause, 559
GUIDs
SQL syntax for NEWID function, 204
SQL syntax for STRTOUUID function, 261
SQL syntax for UUIDTOSTR function, 274
UNIQUEIDENTIFIER data type, 75
gunzip utility
DECOMPRESS function, 148
gzip utility
COMPRESS function, 121

H

handling
errors in embedded SQL, 717
errors in Transact-SQL, 616
HASH function
SQL syntax, 172
hashing
supported algorithms, 172
HAVING clause
search conditions, 20
SELECT statement, 651
HEADER clause
CREATE FUNCTION statement, 401
CREATE PROCEDURE statement, 419
help
technical support, xix
HELP statement
SQL syntax, 562
hexadecimal constants, xi
(see also binary literals)
binary, 173
hexadecimal escape sequences
in SQL strings, 9
hexadecimal strings
about, 173
HEXTOINT function
SQL syntax, 173
histograms
only partially updated by LOAD TABLE, 590
selectivity estimates, 28
SYSCOLSTAT system view, 758
updating with CREATE STATISTICS, 442
HOLDLOCK table hint
FROM clause, 538
host variables
common element in SQL syntax, 295
declaring in embedded SQL, 476
hostvar
common element in SQL syntax, 295
HOUR function
SQL syntax, 174
HOURS function
SQL syntax, 175
how dates are stored
about, 67
HTML_DECODE function
SQL syntax, 176

HTML_ENCODE function
SQL syntax, 177
HTTP
setting headers, 922
setting options, 922, 924
HTTP functions
alphabetical list, 98
HTTP_DECODE function
SQL syntax, 178
HTTP_ENCODE function
SQL syntax, 178
HTTP_HEADER function
SQL syntax, 179
HTTP_VARIABLE function
SQL syntax, 181

I

I/O
recalibrating the I/O cost model, 303
iAnywhere developer community
newsgroups, xix
icons
used in manuals, xvii
identifiers
about, 7
maximum length in SQL Anywhere, 7
SQL syntax, 7
IDENTITY column
@@identity, 41
IDENTITY function
SQL syntax, 182
idle server
creating events using the CREATE EVENT statement, 390
IdleTime event condition
about, 158
IdleTimeout property
setting with sa_server_option, 917
IF expressions
search conditions, 20
SQL syntax, 17
IF statement
SQL syntax, 563
Transact-SQL syntax, 565
IF UPDATE clause
in triggers, 462
in triggers in Transact-SQL, 468
IFNULL function
SQL syntax, 182
image backups
creating using the BACKUP statement, 346
IMAGE data type
syntax, 74
image SQL functions
about, 102
images
reading from the database, 620
importing data
into tables from files, 568
IN search condition
SQL syntax, 25
INCLUDE statement
SQL syntax, 567
independent variables
regression line, 222
index hints
FROM clause, 536
INDEX_ESTIMATE function
SQL syntax, 183
indexes
ALTER INDEX statement, 311
automatically created, 407
built-in functions, 405
clustering using the ALTER INDEX statement, 311
compressing, 628
creating using the CREATE INDEX statement, 405
dropping using the DROP statement, 498
foreign keys, 407
functions, 406
naming, 407
optimizing for OLAP workloads, 406
owner, 407
physical indexes recorded in SYSPHYSIDX system view, 777
primary keys, 407
renaming using the ALTER INDEX statement, 311
system views, 768
table use, 407
unique, 405
unique names, 407
VALIDATE statement, 713
views, 407, 813
virtual, 405
indicator variables

about, 295
indicator-variable
common element in SQL syntax, 295
indicators
comments, 42
initializing
databases using the CREATE DATABASE statement, 374
INNER JOIN clause
FROM clause SQL syntax, 535
INPUT INTO statement (see INPUT statement)
INPUT statement
cannot be used in stored procedures, 571
SQL syntax, 568
INSERT statement
SQL syntax, 573
inserting
BLOBs using the SET statement, 656
multi-row, 515
rows in bulk, 585
rows into tables, 573
rows using cursors, 614
wide inserts, 515
inserting BLOBs
using xp_read_file system procedure, 947
INSERTING condition
triggers, 26
INSERTSTR function
SQL syntax, 184
INSTALL JAVA statement
installing Java classes, 578
SQL syntax, 578
install-dir
documentation usage, xvi
installing
Java classes, 578
INSTEAD OF triggers
CREATE TRIGGER statement, 462
INT data type (see INTEGER data type)
INTEGER data type
syntax, 60
integers
generating table of , 912
integrated logins
REVOKE statement, 636
integrity
constraints in CREATE TABLE statement, 456
Interactive SQL
alphabetical list of all statements, 294
behavior when connecting, 371
BYE statement syntax, 522
CLEAR statement syntax, 362
CONFIGURE statement syntax, 369
CONNECT statement syntax, 370
DESCRIBE statement syntax, 494
DISCONNECT statement syntax, 497
EXIT statement syntax, 522
HELP statement syntax, 562
INPUT statement syntax, 568
OUTPUT statement syntax, 604
PARAMETERS statement syntax, 608
procedure profiling, 914
QUIT statement syntax, 522
READ statement syntax, 618
RESUME statement unsupported, 633
return codes, 522
SET CONNECTION statement syntax, 661
SET OPTION statement syntax, 667
specifying encoding for INPUT statement, 570
specifying encoding for OUTPUT statement, 606
specifying encoding for READ statement, 618
START DATABASE statement, 674
START ENGINE statement syntax, 676
START LOGGING statement syntax, 678
STOP LOGGING statement syntax, 686
SYSTEM statement syntax, 691
INTERSECT statement
SQL syntax, 580
intersecting
result of multiple select statements, 580
Interval event condition
about, 158
INTO clause
INPUT statement, 568
SELECT statement, 650
INTTOHEX function
SQL syntax, 184
invoking
procedures using the CALL statement, 357
IOParallelism property
DB_EXTENDED_PROPERTY function, 143
IS
logical operators description, 12
three-valued logic, 27
IS FALSE search condition
SQL syntax, 26

IS NOT NULL search condition
SQL syntax, 26
IS NULL search condition
SQL syntax, 26
IS TRUE search condition
SQL syntax, 26
IS UNKNOWN search condition
SQL syntax, 26
ISDATE function
SQL syntax, 185
ISNULL function
SQL syntax, 186
ISNUMERIC function
SQL syntax, 186
isolation levels
cursors, 601
table hints, 538
isolation_level option
setting for DELETE statements, 487
setting for EXCEPT statement, 514
setting for INSERT statements, 575
setting for INTERSECT statement, 581
setting for SELECT statements, 653
setting for UNION statement, 696
setting for UPDATE statements, 705
ISYSARTICLE system table
about, 726
ISYSARTICLECOL system table
about, 726
ISYSATTRIBUTE system table
about, 727
ISYSATTRIBUTENAME system table
about, 727
ISYSCAPABILITY system table
about, 727
ISYSCAPABILITYNAME system table
about, 727
ISYSCHECK system table
about, 727
ISYSCOLPERM system table
about, 727
ISYSCOLSTAT system table
about, 727
loading the statistics, 584
ISYSCONSTRAINT system table
about, 727
ISYSDEPENDENCY system table
about, 728
ISYSDOMAIN system table
about, 728
ISYSEVENT system table
about, 728
ISYSEVENTTYPE system table
about, 728
ISYSEXTERNLOGIN system table
about, 728
ISYSFILE system table
about, 728
ISYSFKEY system table
about, 728
ISYSGROUP system table
about, 729
ISYSHISTORY system table
about, 729
ISYSIDX system table
about, 729
ISYSIDXCOL system table
about, 729
ISYSJAR system table
about, 729
ISYSJARCOMPONENT system table
about, 729
ISYSJAVACLASS system table
about, 729
ISYSLOGINMAP system table
about, 730
ISYSMVOPTION system table
about, 730
ISYSMVOPTIONNAME system table
about, 730
ISYSOBJECT system table
about, 730
ISYSOPTION system table
about, 730
ISYSOPTSTAT system table
about, 730
ISYSPHYSIDX system table
about, 730
ISYSPROCEDURE system table
about, 731
ISYSPROCPARM system table
about, 731
ISYSPROCPERM system table
about, 731
ISYSPROXYTAB system table
about, 731

ISYSPUBLICATION system table
about, 731
ISYSREMARK system table
about, 731
ISYSREMOTEOPTION system table
about, 731
ISYSREMOTEOPTIONTYPE system table
about, 731
ISYSREMOTETYPE system table
about, 732
ISYSREMOTEUSER system table
about, 732
ISYSSCHEDULE system table
about, 732
ISYSSERVER system table
about, 732
adding servers, 435
remote servers for Component Integration Services, 435
ISYSSOURCE system table
about, 732
ISYSSQLSERVERTYPE system table
about, 732
ISYSSUBSCRIPTION system table
about, 732
ISYSSYNC system table
about, 732
ISYSSYNCSCRIPT system table
about, 733
ISYSTAB system table
about, 733
ISYSTABCOL system table
about, 733
ISYSTABLEPERM system table
about, 733
ISYSTRIGGER system table
about, 733
ISYSTYPEMAP system table
about, 733
ISYSUSER system table
about, 733
ISYSUSER system tables
ISYSUSER, 733
ISYSUSERAUTHORITY system table
about, 734
ISYSUSERMESSAGE system table
about, 734
ISYSUSERTYPE system table
about, 734
ISYSVIEW system table
about, 734
ISYSWEBSERVICE system table
about, 734
adding servers, 323
adding services, 438
altering services, 323
iterating
over cursors, 530

J

JAR files
installing, 578
removing, 627
Java
converting Java and SQL, 88
installing, 578
system tables, 751
user-defined functions, 96
Java and SQL data type conversion
about, 88
Java classes
CREATE DATABASE statement, 375
loaded in the database, 881
troubleshooting, 881
Java data types
converting from SQL, 89
converting to SQL, 88
Java signatures
CREATE PROCEDURE statement, 418
example, 404
Java to SQL data type conversion
about, 88
Java VM
stopping, 685
jConnect
CREATE DATABASE statement, 379
JCONNECT clause
CREATE DATABASE statement, 379
JDBC
data type conversion, 88
Java to SQL data type conversion, 88
SQL to Java data type conversion, 89
upgrading database components, 301
join operators
compatibility with ASE, 14

joins
ANSI equivalency, 236
deleting rows based on joins, 485
FROM clause SQL syntax, 535
updates, 710
updates based on, 705, 711

K

Kerberos
adding comments using the COMMENT statement, 365
case sensitivity of principals, 551
granting, 548
revoking KERBEROS LOGIN, 636
KEY JOIN clause
FROM clause SQL syntax, 535
keywords
SQL syntax, 4
kind tests
supported by openxml system procedure, 836

L

labels
for statements, 296
statements, 547
language elements
SQL syntax, 4
LANGUAGE JAVA clause
CREATE PROCEDURE statement, 418
large binary objects
getting from columns, 542
large databases
index storage, 406
LAST USER
special value, 32
LAST_VALUE function
SQL syntax, 187
lateral derived tables
FROM clause outer references, 537
LCASE function
SQL syntax, 189
leap years
about, 69
LEAVE statement
SQL syntax, 582
LEFT function
SQL syntax, 190
LEFT OUTER JOIN clause
FROM clause SQL syntax, 535
LENGTH function
SQL syntax, 190
LESSER function
SQL syntax, 191
LIKE search condition
case-sensitivity, 23
collations, 23
maximum pattern length, 23
SQL syntax, 23
limiting the number of rows returned
about, 648
limits, xi
(see also limitations)
LIST function
SQL syntax, 192
lists
LIST function syntax, 192
literal strings (see string literals)
literals
about, 9
LivenessTimeout property
setting with sa_server_option, 917
LOAD STATISTICS statement
SQL syntax, 584
LOAD TABLE statement
SQL syntax, 585
loading
bulk inserts, 585
loading data
multibyte character sets, 587
local temporary tables
creating, 483
creating using the CREATE LOCAL TEMPORARY TABLE statement, 409
local variables
definition, 36
SQL syntax, 36
LOCATE function
SQL syntax, 194
LOCK TABLE statement
SQL syntax, 593
locking
blocks, 850
tables, 593
locks
displaying, 882

types, 884
log files
allocating space using ALTER DBSPACE, 305
analyzing the request log, 872, 873
determining available space, 865
LOG function
SQL syntax, 195
LOG10 function
SQL syntax, 196
LogFreePercent event condition
about, 158
LogFreeSpace event condition
about, 158
logging
START LOGGING statement, 678
STOP LOGGING statement, 686
updating columns without, 722
logical operators
SQL syntax, 12
three-valued logic, 27
logins
assigning for remote servers, 397
dropping for remote servers, 502
LogSize event condition
about, 158
LONG BINARY data type
syntax, 75
LONG BIT VARYING data type (see LONG VARBIT data type)
long column names
retrieving, 491
LONG NVARCHAR data type
describing, 49
syntax, 49
LONG VARBIT data type
syntax, 65
LONG VARCHAR data type
syntax, 50
LOOP statement
SQL syntax, 595
looping
over cursors, 530
LOWER function
SQL syntax, 196
lowercase strings
LCASE function, 189
LOWER function, 196
LTRIM function
SQL syntax, 197

M

MAPI
extended system procedures, 951
return codes, 955
MAPI message type
DROP REMOTE MESSAGE TYPE statement, 504
SQL Remote ALTER REMOTE MESSAGE TYPE statement, 319
SQL Remote CREATE REMOTE MESSAGE TYPE statement, 431
match types
referential integrity, 457
materialized views
ALTER INDEX statement, 311
ALTER MATERIALIZED VIEW statement, 313
CREATE MATERIALIZED VIEW statement, 411
determining status, 887
DROP MATERIALIZED VIEW statement, 498
listing all materialized views in the database, 887
unloading, 700
validating indexes, 713
materialized-view-name
common element in SQL syntax, 295
materialized_view_optimization option
setting for DELETE statements, 486
setting for EXCEPT statements, 513
setting for INSERT statements, 574
setting for INTERSECT statement, 580
setting for SELECT statements, 653
setting for UNION statements, 695
setting for UPDATE statements, 705
mathematical expressions
arithmetic operators, 12
MAX function
SQL syntax, 198
max_query_tasks option
setting for DELETE statements, 487
setting for EXCEPT statement, 514
setting for INSERT statements, 575
setting for INTERSECT statement, 581
setting for SELECT statements, 653
setting for UNION statement, 696
setting for UPDATE statements, 705
maximum

date ranges, 71
memory
allocating for descriptor areas, 299
message control parameters
setting, 668
MESSAGE statement
SQL syntax, 597
messages
creating, 413
displaying, 597
dropping remote types, 504
SQL Remote altering remote types, 319
SQL Remote creating remote types, 431
messages windows
printing messages in, 613
method signatures
Java, 418
migrating databases
sa_migrate system procedure, 888
MIME base64
decoding data, 108
encoding data, 108
MIN function
SQL syntax, 199
minimum
date ranges, 71
MINUTE function
SQL syntax, 199
MINUTES function
SQL syntax, 200
MIRROR clause
CREATE DATABASE statement, 380
MobiLink
ALTER PUBLICATION statement, 317
ALTER SYNCHRONIZATION SUBSCRIPTION statement, 328
ALTER SYNCHRONIZATION USER statement, 330
CREATE PUBLICATION statement, 427
CREATE SYNCHRONIZATION SUBSCRIPTION statement, 445
CREATE SYNCHRONIZATION USER statement, 448
DROP PUBLICATION statement, 503
DROP SYNCHRONIZATION SUBSCRIPTION statement, 510
START SYNCHRONIZATION DELETE statement, 681
STOP SYNCHRONIZATION DELETE statement, 688
MobiLink users
ALTER SYNCHRONIZATION USER statement, 330
CREATE SYNCHRONIZATION USER statement, 448
DROP SYNCHRONIZATION USER statement, 511
MOD function
SQL syntax, 201
MONEY data type
syntax, 64
money data types
about, 64
MONEY, 64
SMALLMONEY, 64
monitoring performance
execution time determination, 873
MONTH function
SQL syntax, 202
MONTHNAME function
SQL syntax, 202
MONTHS function
SQL syntax, 203
multi-row fetches
FETCH statement, 527
OPEN statement, 602
multi-row inserts
about, 515
multibyte character sets
unloading data, 587, 701
multiple result sets
retrieving, 633

N

names
column names, 16
NATIONAL CHAR data type (see NCHAR data type)
NATIONAL CHAR VARYING data type (see NVARCHAR data type)
NATIONAL CHARACTER data type (see NCHAR data type)
NATIONAL CHARACTER VARYING data type (see NVARCHAR data type)
NATURAL JOIN clause
FROM clause SQL syntax, 535

NCHAR COLLATION clause
collation tailoring, 379
CREATE DATABASE statement, 379
NCHAR data type
comparing with CHAR data type, 81
describing, 50
syntax, 50
using DESCRIBE on an NCHAR column, 50
NCHAR function
SQL syntax, 204
NCHAR VARYING data type (see NVARCHAR data type)
NcharCollation property
DB_EXTENDED_PROPERTY function, 143
nesting
user-defined transactions using the BEGIN TRANSACTION statement, 354
new line characters
in SQL strings, 9
NEWID function
SQL syntax, 204
newsgroups
technical support, xix
NEXT_CONNECTION function
SQL syntax, 205
NEXT_DATABASE function
SQL syntax, 207
NEXT_HTTP_HEADER function
SQL syntax, 207
NEXT_HTTP_VARIABLE function
SQL syntax, 208
NEXT_SOAP_HEADER function
SQL syntax, 209
NextScheduleTime property
DB_EXTENDED_PROPERTY function, 143
NO RESULT SET clause
about, 417, 425
NO SCROLL cursors
declaring, 478
NOLOCK table hint
FROM clause, 538
NOT
bitwise operators, 13
logical operators description, 12
three-valued logic, 27
NOT ENCRYPTED clause
ALTER MATERIALIZED VIEW statement, 313
NOT TRANSACTIONAL clause
CREATE TABLE statement, 452
NOW function
SQL syntax, 210
NTEXT data type
syntax, 51
NULL
about, 43
ASE compatibility, 44
DISTINCT clause, 44
ISNULL function, 186
NULL value, 43
set operators, 44
three-valued logic, 27, 43
NULL constants
converting to NUMERIC, 84
converting to string types, 84
NULL values
domains, 386
NULLIF function
about, 210
using with CASE expressions, 18
number
common element in SQL syntax, 295
NUMBER function
SQL syntax, 211
updates, 704, 710
number of rows
system views, 795
numeric constants (see binary literals)
NUMERIC data type
converting from DOUBLE, 86
syntax, 61
numeric data types
about, 56
BIGINT, 56
BIT, 57
converting DOUBLE to NUMERIC, 86
DECIMAL, 57
DOUBLE, 58
FLOAT, 59
INTEGER, 60
NUMERIC, 61
REAL, 62
SMALLINT, 62
TINYINT, 63
numeric functions
alphabetical list, 98
NVARCHAR data type

describing, 52
syntax, 52
using DESCRIBE on an NVARCHAR column, 52

O

ODBC
declaring static cursors, 478
OLAP
CUBE operation, 560
GROUP BY clause, 559
GROUPING function, 171
GROUPING SETS operation, 559
ROLLUP operation, 560
WINDOW clause, 719
OLAP functions
AVG function, 107
COUNT function, 129
COVAR_POP function, 131
CUME_DIST function, 135
DENSE_RANK function, 151
MAX function, 198
MIN function, 199
PERCENT_RANK function, 213
RANK function, 221
REGR_AVGX function, 222
REGR_AVGY function, 223
REGR_COUNT function, 224
REGR_INTERCEPT function, 225
REGR_R2 function, 227
REGR_SLOPE function, 228
REGR_SXX function, 229
REGR_SXY function, 230
ROW_NUMBER function, 240
STDDEV function, 257
STDDEV_POP function, 257
STDDEV_SAMP function, 258
SUM function, 264
VAR_POP function, 275
VAR_SAMP function, 276
ON EXCEPTION RESUME clause
about, 417
ON EXISTING ERROR clause
behavior with DEFAULT columns, 574
ON EXISTING SKIP clause
behavior with DEFAULT columns, 574
ON phrase
search conditions, 20
on_tsql_error option
and ON EXCEPTION RESUME clause, 417
online books
PDF, xii
OPEN statement
SQL syntax, 601
opening cursors
OPEN statement, 601
openxml system procedure
list of supported metaproperties, 835
supported kind tests, 836
syntax, 835
operating systems
executing commands, 691
operator precedence
SQL syntax, 14
operators
about, 11
arithmetic operators, 12
bitwise operators, 13
comparison operators, 11
logical operators description, 12
precedence of operators, 14
string operators, 13
optimization
defining existing tables and, 395
forcing using FORCE OPTIMIZATION option, 486, 513, 574, 580, 653, 695, 705
optimization_goal option
setting for DELETE statements, 487
setting for EXCEPT statement, 514
setting for INSERT statements, 575
setting for INTERSECT statement, 581
setting for SELECT statements, 653
setting for UNION statement, 696
setting for UPDATE statements, 705
optimization_level option
setting for DELETE statements, 487
setting for EXCEPT statement, 514
setting for INSERT statements, 575
setting for INTERSECT statement, 581
setting for SELECT statements, 653
setting for UNION statement, 696
setting for UPDATE statements, 705
optimization_workload option
setting for DELETE statements, 487
setting for EXCEPT statement, 514
setting for INSERT statements, 575

setting for INTERSECT statement, 581
setting for SELECT statements, 653
setting for UNION statement, 696
setting for UPDATE statements, 705
optimizer
CREATE STATISTICS statement, 442
explicit selectivity estimates, 28
optimizer plans
getting text specification, 524
optimizer statistics
dropping using the DROP STATISTICS statement, 508
optimizer tables
about, 735
OPTION clause
DELETE statement, 486
EXCEPT statement, 513
INSERT statement, 574
INTERSECT statement, 580
SELECT statement, 653
UNION statement, 695
UPDATE statement, 705
options
collation tailoring, 376
getting values, 546
initial settings, 936, 945
overriding, 914
quoted_identifier and T-SQL compatibility, 19
setting, 664
setting in Interactive SQL, 369, 667
setting in Transact-SQL, 658
setting remote, 668
setting with sp_tsql_environment system procedure, 945
system views, 776
views, 814, 822
OR
bitwise operators, 13
logical operators description, 12
three-valued logic, 27
Oracle databases
migrating to SQL Anywhere using sa_migrate system procedure, 889
ORDER BY clause
about, 651
SELECT statement, 648
WINDOW clause, 719
order of operations
SQL operator precedence, 14
out of disk space
creating events using the CREATE EVENT statement, 390
outer references
FROM clause, 537
lateral derived tables, 537
OUTPUT statement
SQL syntax, 604
owner
common element in SQL syntax, 296

P

packages
installing Java classes, 578
removing Java classes, 627
PAGE SIZE clause
CREATE DATABASE statement, 380
page sizes
creating databases, 380
page usage
tables, 931
parallel backups
BACKUP statement, 346
parameterized views
about, 471
parameters
Interactive SQL command files, 608
PARAMETERS statement
SQL syntax, 608
PARTITION BY clause
WINDOW clause, 719
passthrough mode
PASSTHROUGH statement (SQL Remote), 609
starting, 609
stopping, 609
PASSTHROUGH statement
SQL syntax, 609
passwords
character set conversion, 549
maximum length, 549
sa_verify_password system procedure, 936
PATINDEX function
SQL syntax, 212
pattern matching
case-sensitivity, 23
collations, 23

LIKE search condition, 23
maximum pattern length, 23
PATINDEX function, 212
PCTFREE setting
ALTER TABLE statement, 332
CREATE LOCAL TEMPORARY TABLE syntax, 409
CREATE TABLE statement, 450
DECLARE LOCAL TEMPORARY TABLE syntax, 483
LOAD TABLE syntax, 585
PDF
documentation, xii
percent sign
comment indicator, 42
PERCENT_RANK function
SQL syntax, 213
performance
compression statistics, 848
pre-allocating space, 305
recalibrating the database server, 301
recalibrating the I/O cost model, 303
updates, 711
permissions
granting, 548
granting ALL, 548
granting ALTER, 548
granting BACKUP, 548
granting CONNECT, 548
granting CONSOLIDATE, 553
granting DBA, 548
granting DELETE, 548
granting EXECUTE, 548
granting GROUP, 548
granting INSERT, 548
granting INTEGRATED LOGIN, 548
granting KERBEROS LOGIN, 548
granting MEMBERSHIP IN GROUP, 548
granting PUBLISH, 555
granting REFERENCES, 548
granting REMOTE, 556
granting REMOTE DBA, 558
granting RESOURCE, 548
granting SELECT, 548
granting UPDATE, 548
granting VALIDATE, 548
revoking, 636
revoking ALL, 636
revoking ALTER, 636
revoking BACKUP, 636
revoking CONNECT, 636
revoking CONSOLIDATE, 638
revoking DBA, 636
revoking DELETE, 636
revoking EXECUTE, 636
revoking GROUP, 636
revoking INSERT, 636
revoking INTEGRATED LOGIN, 636
revoking KERBEROS LOGIN, 636
revoking MEMBERSHIP IN GROUP, 636
revoking PUBLISH, 639
revoking REFERENCES, 636
revoking REMOTE, 640
revoking REMOTE DBA, 641
revoking RESOURCE, 636
revoking SELECT, 636
revoking UPDATE, 636
revoking VALIDATE, 636
SYSCOLAUTH view, 810
SYSTABAUTH consolidated view, 821
system views, 757, 799
physical indexes
recorded in SYSPHYSIDX system view, 777
PI function
SQL syntax, 214
PLAN function
SQL syntax, 214
plans
and cursors, 163, 169, 214, 269
getting text specification, 524
SQL syntax, 163, 169, 214, 269
pooling
enabling connection pooling, 671
population covariance
about, 131
population variance
about, 275
positioned DELETE statement
SQL syntax, 488
POWER function
SQL syntax, 215
precedence
SQL operator precedence, 14
PRECEDING clause
WINDOW clause, 720
predicates

ALL, ANY, and SOME search conditions, 21
BETWEEN search condition, 22
comparison operators, 11
EXISTS search condition, 25
explicit selectivity estimates, 28
IN search condition, 25
IS NOT NULL search condition, 26
IS NULL search condition, 26
IS TRUE or FALSE search conditions, 26
IS UNKNOWN search condition, 26
LIKE search condition, 23
SQL subqueries in, 21
SQL syntax, 20
three-valued logic, 27
PREPARE statement
SQL syntax, 610
PREPARE TO COMMIT statement
SQL syntax, 612
prepared statements
dropping using the DROP STATEMENT statement, 507
executing, 515
preparing
for two-phase commit, 612
statements, 610
primary keys
ALTER INDEX statement, 311
clustering using the ALTER INDEX statement, 311
generating unique values, 204
generating unique values using UUIDs, 204
integrity constraints in CREATE TABLE statement, 456
order of columns in CREATE TABLE statement, 456
remote tables, 938, 940
renaming using the ALTER INDEX statement, 311
UUIDs and GUIDs, 204
primary tables
system views, 765
PRINT statement
Transact-SQL syntax, 613
printing
messages in the message window, 613
procedure parameters
listing in Interactive SQL, 494
procedure profiling
disabling from Interactive SQL, 917
enabling from Interactive SQL, 917
in Interactive SQL, 914
sa_procedure_profile system procedure, 902
summary of procedures, 904
viewing in Interactive SQL, 904
ProcedureProfiling
setting with sa_server_option, 917
procedures
alphabetical list , 835
alphabetical list of system procedures, 835
altering using the ALTER PROCEDURE statement, 315
CREATE PROCEDURE SQL statement, 425
creating, 414
creating in Transact-SQL, 425
dropping using the DROP statement, 498
executing in dynamic SQL, 519
executing stored in Transact-SQL, 517
exiting, 634
extended list, 951
external function calls, 400, 418
invoking using the CALL statement, 357
raising errors in Transact-SQL, 616
replicating using the ALTER PROCEDURE statement, 315
resuming execution of, 633
returning values from, 634
selecting from, 536
system, 833
Transact-SQL list, 962
variable result sets, 416, 491, 610
product name
retrieving, 960
ProfileFilterConn property
setting with sa_server_option, 917
ProfileFilterUser property
setting with sa_server_option, 917
properties
CONNECTION_PROPERTY function, 123
DB_PROPERTY function, 147
PROPERTY function, 216
server, 216
Properties property
DB_EXTENDED_PROPERTY function, 143
PROPERTY function
SQL syntax, 216
PROPERTY_DESCRIPTION function
SQL syntax, 217

PROPERTY_NAME function
SQL syntax, 217
PROPERTY_NUMBER function
SQL syntax, 218
proxy procedures
creating, 414
proxy tables
CREATE TABLE statement, 452
creating using the CREATE EXISTING TABLE statement, 395
publications
ALTER PUBLICATION statement, 317
CREATE PUBLICATION statement, 427
DROP PUBLICATION statement, 503
UPDATE statement, 706
UPDATE statement (SQL Remote), 711
publish permissions
granting, 555
revoking, 639
publisher
address, 504
GRANT PUBLISH statement, 555
remote, 556
SQL Remote address, 431
SQL Remote addresses, 319
PunctuationSensitivity property
DB_EXTENDED_PROPERTY function, 143
PURGE clause
FETCH statement, 527
PUT statement
SQL syntax, 614
putting
rows into cursors, 614

Q

QUARTER function
SQL syntax, 218
query-block
common element in SQL syntax, 296
query-expression
common element in SQL syntax, 296
QUIT statement
SQL syntax, 522
quitting
Interactive SQL, 522
QuittingTime property
setting with sa_server_option, 917
quotation marks
compatibility with ASE, 19
database objects, 7
single vs. double, 19
SQL identifiers, 7
QUOTE option
LOAD TABLE statement, 588
UNLOAD TABLE statement, 701
quoted_identifier option
setting with Transact-SQL SET statement, 658
T-SQL expression compatibility, 19
quotes, xi
(see also quotation marks)
QUOTES option
LOAD TABLE statement, 587, 588
UNLOAD TABLE statement, 701

R

R-squared
regression lines, 227
RADIANS function
SQL syntax, 219
RAISERROR statement
Transact-SQL syntax, 616
raising
errors in Transact-SQL, 616
RAND function
SQL syntax, 219
random numbers
RAND function, 219
range
date type, 71
RANGE clause
WINDOW clause, 719
RANK function
SQL syntax, 221
ranking functions
alphabetical list, 94
CUME_DIST function, 135
DENSE_RANK function, 151
PERCENT_RANK function, 213
RANK function, 221
re-describing cursors
CREATE PROCEDURE statement, 417
read committed
FROM clause, 538
read only

locking tables, 593
READ statement
SQL syntax, 618
read uncommitted
FROM clause, 538
READCOMMITTED table hint
FROM clause, 538
reading
text and image values from the database, 620
reading files
stored procedures, 947, 949
reading SQL statements from files
about, 618
READPAST table hint
FROM clause, 538
READTEXT statement
Transact-SQL syntax, 620
READUNCOMMITTED table hint
FROM clause, 538
REAL data type
syntax, 62
recalibrating cost models
about, 301
recovery
LOAD TABLE, 588
referential integrity
actions, 459
FROM clause, 536
match clause in CREATE TABLE statement, 457
REFRESH MATERIALIZED VIEW statement
SQL syntax, 621
REFRESH TRACING LEVEL statement
diagnostic tracing, 623
SQL syntax, 623
REGR_AVGX function
SQL syntax, 222
REGR_AVGY function
SQL syntax, 223
REGR_COUNT function
SQL syntax, 224
REGR_INTERCEPT function
SQL syntax, 225
REGR_R2 function
SQL syntax, 227
REGR_SLOPE function
SQL syntax, 228
REGR_SXX function
SQL syntax, 229
REGR_SXY function
SQL syntax, 230
REGR_SYY function
SQL syntax, 231
regression functions
REGR_AVGX function, 222
REGR_AVGY function, 223
REGR_COUNT function, 224
REGR_INTERCEPT function, 225
REGR_R2 function, 227
REGR_SLOPE function, 228
REGR_SXX function, 229
REGR_SXY function, 230
REGR_SYY function, 231
relationships
system views, 765
RELEASE SAVEPOINT statement
SQL syntax, 625
releasing
savepoints, 625
REMAINDER function
SQL syntax, 233
RememberLastPlan property
setting with sa_server_option, 918
RememberLastStatement property
setting with sa_server_option, 918
remote data access
disconnecting, 321
FORWARD TO statement, 533
remote DBA permissions
granting, 558
revoking, 641
remote message types
dropping, 504
SQL Remote altering, 319
SQL Remote creating, 431
remote options
SET REMOTE OPTION statement (SQL Remote), 668
remote permissions
granting, 556
revoking, 640
remote procedures
creating, 414, 417
creating in Transact SQL, 425
REMOTE RESET statement
SQL syntax, 626
remote servers

altering attributes using the ALTER SERVER statement, 321
assigning logins for, 397
capabilities, 755, 944
CREATE SERVER statement, 435
CREATE TABLE statement, 450
disconnecting, 321
dropping logins for remote servers, 502
dropping using the DROP SERVER statement, 505
sending SQL statements to, 533
SYSCAPABILITYNAME system view, 756
remote tables
columns, 937
CREATE TABLE statement, 452
foreign keys, 938, 940
listing, 942
primary keys, 938, 940
remote users
REVOKE REMOTE statement, 640
remoteoption view
about, 784
remoteoptiontype view
about, 785
REMOVE JAVA statement
SQL syntax, 627
removing
Java classes, 627
permissions, 636
renaming
columns, 337
columns using the ALTER TABLE statement, 332
constraints, 337
tables, 337
tables using the ALTER TABLE statement, 332
REORGANIZE TABLE statement
SQL syntax, 628
reorganizing tables
REORGANIZE TABLE, 628
REPEAT function
SQL syntax, 233
repeatable reads
FROM clause, 538
REPEATABLEREAD table hint
FROM clause, 538
REPLACE function
SQL syntax, 234
replacing objects
sa_make_object, 885
REPLICATE function
SQL syntax, 235
replication
ALTER TABLE statement, 332
procedures using the ALTER PROCEDURE statement, 315
request logging
analyzing the request log with sa_get_request_profile, 872
analyzing the request log with sa_get_request_times, 873
enabling from Interactive SQL, 919
request timing
sa_performance_diagnostics system procedure, 897
RequestFilterConn property
setting with sa_server_option, 918
RequestFilterDB property
setting with sa_server_option, 919
RequestLogFile property
setting with sa_server_option, 919
RequestLogging property
setting with sa_server_option, 919
RequestLogMaxSize property
setting with sa_server_option, 920
RequestLogNumFiles property
setting with sa_server_option, 920
requests
obtaining timing information, 897
RequestTiming property
setting with sa_server_option, 920
reserved words
SQL syntax, 4
using as identifiers, 19
RESIGNAL statement
SQL syntax, 630
resignaling
exceptions, 630
resolving conflicts
CONFLICT function for SQL Remote, 123
RESTORE DATABASE statement
SQL syntax, 631
restoring
databases from archives, 631
result sets
resuming execution of procedures, 633
retrieving multiple result sets, 633
selecting from stored procedures, 536
shape of, 491

unloading, 698
variable, 416, 491, 610
RESUME statement
not supported in Interactive SQL, 633
SQL syntax, 633
resuming
execution of procedures, 633
retrieving
long column names, 491
multiple result sets, 633
retrieving dates and times from the database
about, 68
return codes
EXIT statement [Interactive SQL], 522
RETURN statement
SQL syntax, 634
returning
values from procedures, 634
REVERSE function
SQL syntax, 236
REVOKE BACKUP statement
SQL syntax, 636
REVOKE CONNECT statement
SQL syntax, 636
REVOKE CONSOLIDATE statement
SQL syntax, 638
REVOKE DBA statement
SQL syntax, 636
REVOKE GROUP statement
SQL syntax, 636
REVOKE INTEGRATED LOGIN statement
SQL syntax, 636
REVOKE KERBEROS LOGIN statement
SQL syntax, 636
REVOKE MEMBERSHIP IN GROUP statement
SQL syntax, 636
REVOKE PUBLISH statement
SQL syntax, 639
REVOKE REMOTE DBA statement
SQL syntax, 641
REVOKE REMOTE statement
SQL syntax, 640
REVOKE RESOURCE statement
SQL syntax, 636
REVOKE statement
SQL syntax, 636
REVOKE VALIDATE statement
SQL syntax, 636
revoking
CONSOLIDATE permissions, 638
PUBLISH permissions, 639
remote DBA permissions, 641
REMOTE permissions, 640
REVOKE statement, 636
REWRITE function
SQL syntax, 236
RIGHT function
SQL syntax, 238
RIGHT OUTER JOIN clause
FROM clause SQL syntax, 535
role names
foreign keys in CREATE TABLE statement, 457
role-name
common element in SQL syntax, 296
ROLLBACK statement
SQL syntax, 642
ROLLBACK TO SAVEPOINT statement
SQL syntax, 643
ROLLBACK TRANSACTION statement
Transact-SQL syntax, 644
ROLLBACK TRIGGER statement
SQL syntax, 645
rolling back
transactions, 642, 644, 646
transactions to savepoints, 643
triggers, 645
ROLLUP operation
GROUP BY clause, 560
GROUPING function, 171
WITH ROLLUP clause, 560
ROUND function
SQL syntax, 239
roundoff errors
about, 56
Row Constructor algorithm
DUMMY system table, 726
ROW DELIMITED BY option
LOAD TABLE statement, 588
UNLOAD TABLE statement, 701
row generator
RowGenerator table (dbo), 751
sa_rowgenerator system procedure, 910
row limits
about, 648
row-level triggers
about, 464

ROW_NUMBER function
SQL syntax, 240
rowcount option
setting with Transact-SQL SET statement, 658
RowGenerator system table
about, 751
ROWID function
SQL syntax, 239
rows
deleting all from a table, 693
deleting from cursors, 488
deleting from databases, 485
fetching from cursors, 526
inserting in bulk, 585
inserting into tables, 573
inserting using cursors, 614
limiting number returned, 648
selecting, 648
unloading, 698
updating, 703
ROWS clause
WINDOW clause, 719
RTRIM function
SQL syntax, 242
rules
SQL language syntax, 4

S

sa_ansi_standard_packages system procedure
about, 839
sa_audit_string system procedure
syntax, 841
sa_check_commit system procedure
syntax, 841
sa_clean_database system procedure
syntax, 842
sa_column_stats system procedure
syntax, 845
sa_conn_activity system procedure
syntax, 847
sa_conn_compression_info system procedure
syntax, 848
sa_conn_info system procedure
syntax, 850
sa_conn_list system procedure
syntax, 852
sa_conn_options system procedure
syntax, 853
sa_conn_properties system procedure
syntax, 854
sa_convert_ml_progress_to_timestamp system procedure
syntax, 855
sa_convert_timestamp_to_ml_progress system procedure
syntax, 856
sa_db_info system procedure
syntax, 856
sa_db_list system procedure
syntax, 858
sa_db_properties system procedure
syntax, 858
sa_dependent_views system procedure
syntax, 859
sa_describe_query system procedure
syntax, 860
sa_diagnostic_auxiliary_catalog table
about, 735
sa_diagnostic_blocking table
about, 736
sa_diagnostic_cachecontents table
about, 737
sa_diagnostic_connection table
about, 738
sa_diagnostic_cursor table
about, 739
sa_diagnostic_deadlock table
about, 740
sa_diagnostic_hostvariable table
about, 741
sa_diagnostic_internalvariable table
about, 742
sa_diagnostic_query table
about, 743
sa_diagnostic_request table
about, 745
sa_diagnostic_statement table
about, 747
sa_diagnostic_statistics table
about, 747
sa_diagnostic_tracing_level table
about, 748
sa_disable_auditing_type system procedure
syntax, 864
sa_disk_free_space system procedure

syntax, 865
sa_enable_auditing_type system procedure
syntax, 866
sa_eng_properties system procedure
syntax, 867
sa_flush_cache system procedure
syntax, 868
sa_flush_statistics system procedure
syntax, 868
sa_get_bits system procedure
syntax, 869
sa_get_dtt system procedure
syntax, 870
sa_get_histogram system procedure
syntax, 871
sa_get_request_profile system procedure
syntax, 872
sa_get_request_times system procedure
syntax, 873
sa_get_server_messages system procedure
syntax, 875
sa_http_header_info system procedure
syntax, 876
sa_http_variable_info system procedure
syntax, 876
sa_index_density system procedure
syntax, 877
sa_index_levels system procedure
syntax, 879
sa_java_loaded_classes system procedure
syntax, 881
sa_load_cost_model system procedure
syntax, 881
sa_locks system procedure
syntax, 882
sa_make_object system procedure
syntax, 885
sa_materialized_view_info system procedure
syntax, 887
sa_migrate system procedure
syntax, 888
sa_migrate_create_fks system procedure
syntax, 890
sa_migrate_create_remote_fks_list system procedure
syntax, 892
sa_migrate_create_remote_table_list system procedure
syntax, 893
sa_migrate_create_tables system procedure
syntax, 894
sa_migrate_data system procedure
syntax, 895
sa_migrate_drop_proxy_tables system procedure
syntax, 896
sa_performance_diagnostics system procedure
syntax, 897
sa_performance_statistics system procedure
syntax, 901
sa_procedure_profile system procedure
syntax, 902
sa_procedure_profile_summary system procedure
syntax, 904
sa_recompile_views system procedure
syntax, 906
sa_refresh_materialized_views system procedure
syntax, 907
sa_remove_tracing_data system procedure
syntax, 908
sa_report_deadlocks system procedure
syntax, 908
sa_reset_identity system procedure
syntax, 909
sa_rowgenerator system procedure
syntax, 910
sa_save_trace_data system procedure
syntax, 912
sa_send_udp system procedure
syntax, 913
sa_server_option system procedure
syntax, 914
sa_set_http_header system procedure
syntax, 922
sa_set_http_option system procedure
syntax, 922
sa_set_soap_header system procedure
syntax, 924
sa_set_tracing_level system procedure
syntax, 925
sa_snapshots system procedure
syntax, 926
sa_split_list system procedure
syntax, 927
sa_statement_text system procedure
syntax, 929
sa_table_fragmentation system procedure
syntax, 930
sa_table_page_usage system procedure

syntax, 931
sa_table_stats system procedure
syntax, 931
sa_transactions system procedure
syntax, 932
sa_unload_cost_model system procedure
syntax, 933
sa_validate system procedure
syntax, 934
sa_verify_password system procedure
syntax, 936
sample covariance
about, 132
sample variance
about, 276
samples-dir
documentation usage, xvi
SAVE TRANSACTION statement
Transact-SQL syntax, 646
SAVEPOINT statement
SQL syntax, 647
savepoint-name
common element in SQL syntax, 296
savepoints
creating, 647
releasing, 625
rolling back to savepoints, 643
scheduled events
triggering, 692
WAITFOR statement, 715
scheduling
creating events using the CREATE EVENT statement, 390
events using the ALTER EVENT statement, 308
events using the CREATE EVENT statement, 390
WAITFOR, 715
schemas
creating, 433
default system views, 753
system tables, 726
scripted upload
CREATE PUBLICATION syntax, 427
SCROLL cursors
declaring, 478
search conditions
about, 20
ALL, ANY, and SOME, 21
BETWEEN, 22
EXISTS, 25
explicit selectivity estimates, 28
IN, 25
IS NOT NULL, 26
IS NULL, 26
IS TRUE or FALSE search conditions, 26
IS UNKNOWN search condition, 26
LIKE, 23
SQL syntax, 20
subqueries in, 21
three-valued logic, 27
truth value, 26
search-condition
common element in SQL syntax, 296
SECOND function
SQL syntax, 242
SECONDS function
SQL syntax, 243
secured features
changing with sa_server_option, 921
SecureFeatures property
setting with sa_server_option, 921
security
replication, 558, 641
select list
describing cursors, 490
SELECT statement
selecting from stored procedures, 536
SQL syntax, 648
selecting
for unloading, 698
forming intersections, 580
forming set differences, 513
forming unions, 695
rows, 648
selectivity estimates
source of estimates, 156
user-defined, 28
self_recursion option
setting with Transact-SQL SET statement, 658
SEND AT clause
about, 553, 556
publish, 555
SEND EVERY clause
about, 553, 556
sending
SQL statements to remote servers, 533
sending dates and times to the database

about, 67
serializable
FROM clause, 538
SERIALIZABLE table hint
FROM clause, 538
servers
altering remote attributes using the ALTER SERVER statement, 321
altering web services using the ALTER SERVICE statement, 323
creating, 435
creating events for idle servers using the CREATE EVENT statement, 390
creating web, 438
dropping remote servers , 505
dropping web servers using the DROP SERVICE statement, 506
starting database, 676
stopping database, 684
services
adding comments using the COMMENT statement, 365
altering web services using the ALTER SERVICE statement, 323
creating web, 438
dropping web services using the DROP SERVICE statement, 506
SET CONNECTION statement
SQL syntax, 661
SET DESCRIPTOR statement
SQL syntax, 662
set operators
NULL, 44
SET OPTION statement
embedded SQL syntax, 664
Interactive SQL syntax, 667
SQL syntax, 664
Transact-SQL syntax, 658
SET PARTNER FAILOVER clause
ALTER DATABASE statement, 302
SET PERMANENT statement
Interactive SQL syntax, 667
SET REMOTE OPTION statement
SQL syntax, 668
SET SESSION AUTHORIZATION statement
SQL syntax, 671
SET SQLCA statement
SQL syntax, 670
SET statement
SQL syntax, 656
Transact-SQL syntax, 658
SET TEMPORARY OPTION statement
embedded SQL syntax, 664
Interactive SQL syntax, 667
SQL syntax, 664
SET_BIT function
SQL syntax, 244
SET_BITS function
SQL syntax, 245
setting
connections, 661
descriptor areas, 662
options, 664
options in Interactive SQL, 369, 667
options in Transact-SQL, 658
remote options, 668
SQLCAs, 670
users, 671
values of SQL variables, 656
SETUSER statement
SQL syntax, 671
SHARE BY ALL clause
CREATE TABLE statement, 452
SHORT_PLAN function
SQL syntax, 246
shutting down
databases, 683
SIGN function
SQL syntax, 246
SIGNAL statement
SQL syntax, 673
signaling
errors, 616, 673
exceptions, 630
signatures
Java methods, 418
Java signature example, 404
SIMILAR function
SQL syntax, 247
SIN function
SQL syntax, 248
SKIP option
LOAD TABLE statement, 588
slash-asterisk
comment indicator, 42
slope

regression lines, 228
SMALLDATETIME data type
syntax, 72
SMALLINT data type
syntax, 62
SMALLMONEY data type
syntax, 64
SMTP
extended system procedures, 951
return codes, 959
snapshot isolation
sa_snapshots system procedure, 926
sa_transactions system procedure, 932
SOAP functions
alphabetical list, 98
SOAP services
data typing, 439
SOAP_HEADER function
SQL syntax, 248
SOAPHEADER clause
CREATE FUNCTION statement, 401
CREATE PROCEDURE statement, 420
SOME search condition
SQL syntax, 21
sort keys
generating using the SORTKEY function, 249
sorting
SORTKEY function, 249
sortkey files
generating using the SORTKEY function, 249
SORTKEY function
collation tailoring, 249
SQL syntax, 249
SOUNDEX function
SQL syntax, 253
SP
statement indicators, 298
sp_addgroup system procedure
about, 962
sp_addlogin system procedure
about, 962
sp_addmessage system procedure
about, 413, 962
sp_addtype system procedure
about, 962
sp_adduser system procedure
about, 962
sp_changegroup system procedure
about, 962
sp_column_privileges catalog procedure
about, 963
sp_columns catalog procedure
about, 963
sp_dropgroup system procedure
about, 962
sp_droplogin system procedure
about, 962
sp_dropmessage system procedure
about, 962
sp_droptype system procedure
about, 962
sp_dropuser system procedure
about, 962
sp_fkeys catalog procedure
about, 963
sp_getmessage system procedure
about, 962
sp_helptext system procedure
about, 962
sp_login_environment system procedure
syntax, 936
sp_password system procedure
about, 962
sp_pkeys catalog procedure
about, 963
sp_remote_columns system procedure
syntax, 937
sp_remote_exported_keys system procedure
syntax, 938
sp_remote_imported_keys system procedure
syntax, 940
sp_remote_primary_keys system procedure
syntax, 941
sp_remote_tables system procedure
syntax, 942
sp_servercaps system procedure
syntax, 944
sp_special_columns catalog procedure
about, 963
sp_sproc_columns catalog procedure
about, 963
sp_statistics catalog procedure
about, 963
sp_stored_procedures catalog procedure
about, 963
sp_tables catalog procedure

about, 963
sp_tsql_environment system procedure
syntax, 945
SPACE function
SQL syntax, 254
special characters
SQL strings, 9
used in binary, 9
used in strings , 9
special tables
about, 726
special values
CURRENT DATABASE, 30
CURRENT DATE, 30
CURRENT PUBLISHER, 30
CURRENT TIME, 31
CURRENT TIMESTAMP, 31
CURRENT USER, 32
CURRENT UTC TIMESTAMP, 32
CURRENT_TIMESTAMP, 31
CURRENT_USER, 32
LAST USER, 32
NULL, 43
SQL syntax, 30
SQLCODE, 33
SQLSTATE, 33
TIMESTAMP, 33
USER, 34
UTC TIMESTAMP, 35
special views
about, 753
special-value
common element in SQL syntax, 296
Specification property
DB_EXTENDED_PROPERTY function, 143
SQL
alphabetical list of all statements, 294
SQL Anywhere
documentation, xii
SQL descriptor area
INCLUDE statement, 567
inserting rows using cursors, 614
SQL descriptor areas
DESCRIBE statement, 490
SQL Flagger
SQLFLAGGER function, 255
testing a SQL statement for non-core extensions, 839
SQL functions
aggregate, 93
bit array, 94
data type conversion, 94
date and time, 94
HTTP, 98
image, 102
introduction, 91
miscellaneous, 97
numeric, 98
ranking, 94
SOAP, 98
string, 99
system, 100
text, 102
types of functions, 93
user-defined, 96
SQL language elements
about, 3
SQL Remote
articles SYSARTICLE, 754
articles SYSARTICLECOL, 755
consolidated views, 815, 816, 817
creating subscriptions, 443
setting remote options, 668
system views, 754, 755, 783, 784, 785, 786
SQL Remote system views
article system view, 754
SYSARTICLECOL, 755
SYSPUBLICATION system view, 783
SYSPUBLICATIONS consolidated view, 815
SYSREMOTEOPTION, 784
SYSREMOTEOPTIONS consolidated view, 816
SYSREMOTEOPTIONTYPE, 785
SYSREMOTETYPES consolidated view, 817
SYSREMOTEUSER , 786
SYSREMOTEUSERS consolidated view, 817
SQL Server
migrating to SQL Anywhere using sa_migrate system procedure, 889
SQL statements
alphabetical list of all statements, 294
documentation conventions, 295
installing Java classes, 578
sending to remote servers, 533
SQL syntax
ALL search condition, 21
alphabetical list of all functions, 103

alphabetical list of all statements, 294
alphabetical list of system procedures, 835
ANY search condition, 21
arithmetic operators, 12
BETWEEN search condition, 22
bitwise operators, 13
CASE expression, 17
column names, 16
comments, 42
comparison operators, 11
connection-level variables, 37
constants , 9
constants in expressions, 16
CURRENT DATABASE special value, 30
CURRENT DATE special value, 30
CURRENT PUBLISHER special value, 30
CURRENT TIME special value, 31
CURRENT TIMESTAMP special value, 31
CURRENT USER special value, 32
CURRENT UTC TIMESTAMP special value, 32
CURRENT_TIMESTAMP special value, 31
CURRENT_USER special value, 32
documentation conventions, 295
EXISTS search condition, 25
expressions, 15
functions, 93
identifiers, 7
IF expressions, 17
IN search condition, 25
IS NOT NULL search condition, 26
IS NULL search condition, 26
IS TRUE or FALSE search condition, 26
keywords, 4
LAST USER special value, 32
LIKE search condition, 23
local variables, 36
logical operators, 12
NULL value, 43
operator precedence, 14
operators, 11
predicates, 20
reserved words, 4
search conditions, 20
SOME search condition, 21
special values, 30
SQLCODE special value, 33
SQLSTATE special value, 33
string operators, 13
strings, 8
subqueries, 16
subqueries in search conditions, 21
three-valued logic, 27
TIMESTAMP special value, 33
Transact-SQL expression compatibility, 19
USER special value, 34
UTC TIMESTAMP special value, 35
variables, 36
SQL to Java data type conversion
about, 89
SQL variables
creating, 469
declaring, 477
dropping using the DROP VARIABLE statement, 512
setting values, 656
SQLCA
INCLUDE statement, 567
SQLCAs
setting, 670
SQLCODE
special value, 33
SQLDA
allocating memory for, 299
deallocating, 475
DESCRIBE SQL statement, 490
EXECUTE SQL statement, 515
getting information from, 544
INCLUDE statement, 567
inserting rows using cursors, 614
setting, 662
UPDATE (positioned) statement, 708
SQLDIALECT function
SQL syntax, 255
SQLFLAGGER function
SQL syntax, 255
SQLSetConnectAttr
using with MESSAGE TO CLIENT, 599
SQLSTATE
special value, 33
SQRT function
SQL syntax, 256
square brackets
database objects, 7
SQL identifiers, 7
square root function
SQRT function, 256

standard deviation
STDDEV function, 257
STDDEV_POP function, 257
STDDEV_SAMP function, 258
START AT clause
SELECT statement, 648
START DATABASE statement
SQL syntax, 674
START ENGINE statement
Interactive SQL syntax, 676
START JAVA statement
SQL syntax, 677
START LOGGING statement
Interactive SQL syntax, 678
START SUBSCRIPTION statement
SQL syntax, 679
START SYNCHRONIZATION DELETE statement
SQL syntax, 681
starting
creating events using the CREATE EVENT statement, 390
database servers, 676
databases, 674
Java VM using the START JAVA statement, 677
logging in Interactive SQL, 678
passthrough mode, 609
subscriptions, 679
subscriptions during database extraction, 626
statement applicability indicators
about, 298
statement label
common element in SQL syntax, 296
statement labels
GOTO Transact-SQL statement, 547
statement syntax
alphabetical list of all statements, 294
documentation conventions, 295
statement-level triggers
about, 464
statements
alphabetical list of all statements, 294
dropping prepared statements, 507
executing prepared, 515
GROUP BY clause, 559
grouping in the BEGIN statement, 351
preparing, 610
static cursors
declaring, 478
statistics
CREATE STATISTICS statement, 442
dropping using the DROP STATISTICS statement, 508
flushing to disk, 868
loading, 584
only partially updated by LOAD TABLE, 590
retrieve using sa_get_histogram system procedure, 871
SYSCOLSTAT system view, 758
updating using the ALTER SERVICE statement, 327
STDDEV function
SQL syntax, 257
STDDEV_POP function
SQL syntax, 257
STDDEV_SAMP function
SQL syntax, 258
STOP DATABASE statement
SQL syntax, 683
STOP ENGINE statement
SQL syntax, 684
STOP JAVA statement
SQL syntax, 685
STOP LOGGING statement
Interactive SQL syntax, 686
STOP SUBSCRIPTION statement
SQL syntax, 687
STOP SYNCHRONIZATION DELETE statement
SQL syntax, 688
stopping
database servers, 684
Java VM, 685
logging in Interactive SQL, 686
passthrough mode, 609
stopping databases
STOP DATABASE statement, 683
stopping subscriptions
STOP SUBSCRIPTION statement, 687
stored procedures
converting T-SQL, 278
creating, 414
creating in Transact SQL, 425
executing in dynamic SQL, 519
executing in Transact-SQL, 517
external function calls, 400, 418
INPUT statement cannot be used, 571
selecting from, 536

system procedures, 833
STR function
SQL syntax, 259
string constants (see string literals)
STRING function
SQL syntax, 260
string functions
alphabetical list, 99
string length
LENGTH function, 190
string literals
about, 9
escape sequences, 9
special characters, 9
string operators
SQL syntax, 13
string position
LOCATION function, 194
string-expression
common element in SQL syntax, 296
string_rtruncation option
setting with Transact-SQL SET statement, 658
strings
about, 8
ambiguous conversions to dates, 84, 86
changing the interpretation of delimited strings, 19
converting to dates, 68
delimiter, 19
escape characters, 9
quotation marks, 19
removing trailing blanks , 242
replacing, 234
SQL functions, 99
Transact-SQL, 19
STRIP option
LOAD TABLE statement, 588
strong encryption
CREATE DATABASE statement, 378
STRTOUUID function
SQL syntax, 261
STUFF function
SQL syntax, 262
su
setting users, 671
subqueries
in SQL search conditions, 21
SQL syntax, 16
SUBSCRIBE BY clause
CREATE PUBLICATION statement, 427
subscriptions
ALTER SYNCHRONIZATION SUBSCRIPTION statement, 328
CREATE SUBSCRIPTION statement (SQL Remote), 443
CREATE SYNCHRONIZATION SUBSCRIPTION statement, 445
DROP SUBSCRIPTION statement, 509
DROP SYNCHRONIZATION SUBSCRIPTION statement, 510
REMOTE RESET statement (SQL Remote), 626
START SUBSCRIPTION statement (SQL Remote), 679
STOP SUBSCRIPTION statement (SQL Remote), 687
SYNCHRONIZE SUBSCRIPTION statement (SQL Remote), 689
UPDATE statement, 706
UPDATE statement (SQL Remote), 711
substitution characters
about, 81
comparisons between CHAR and NCHAR, 81
different from character set to character set, 81
SUBSTR function
SQL syntax, 262
SUBSTRING function
SQL syntax, 262
substrings
about, 262
replacing, 234
SUM function
SQL syntax, 264
super types
about, 80
support
newsgroups, xix
SYNCHRONIZE SUBSCRIPTION statement
SQL syntax, 689
synchronizing subscriptions
SYNCHRONIZE SUBSCRIPTION statement (SQL Remote), 689
syntax
arithmetic operators, 12
bitwise operators, 13
CASE expression, 17
column names, 16
comments, 42

comparison operators, 11
connection-level variables, 37
constants , 9
constants in expressions, 16
conventions, 297
CURRENT DATABASE special value, 30
CURRENT DATE special value, 30
CURRENT PUBLISHER special value, 30
CURRENT TIMESTAMP special value, 31
CURRENT USER special value, 32
CURRENT UTC TIMESTAMP special value, 32
CURRENT_TIMESTAMP special value, 31
CURRENT_USER special value, 32
documentation conventions, 295
IF expressions, 17
IS NULL search condition, 26
IS TRUE or FALSE search condition, 26
LAST USER special value, 32
local variables, 36
logical operators, 12
NULL value, 43
predicates, 20
search conditions, 20
special values, 30
SQL CURRENT TIME special value, 31
SQL expressions, 15
SQL functions, 93
SQL identifiers, 7
SQL keywords, 4
SQL operator precedence, 14
SQL operators, 11
SQL reserved words, 4
SQL statements, 294
SQL subqueries, 16
SQL subqueries in search conditions, 21
SQL variables, 36
SQLCODE special value, 33
SQLSTATE special value, 33
string operators, 13
strings, 8
three-valued logic, 27
TIMESTAMP special value, 33
Transact-SQL expression compatibility, 19
USER special value, 34
UTC TIMESTAMP special value, 35
syntax conventions
SQL statements, 297
SYS
default system views, 753
system tables, 726
SYSARTICLE
system view, 754
SYSARTICLECOL
system view, 755
SYSARTICLECOLS
consolidated view, 809
SYSARTICLES
consolidated view, 809
SYSCAPABILITIES
consolidated view, 810
SYSCAPABILITY
system view, 755
SYSCAPABILITYNAME
system view, 756
SYSCATALOG
consolidated view, 810
SYSCHECK
system view, 757
SYSCOLAUTH
consolidated view, 810
SYSCOLLATION
about, 824
SYSCOLLATIONMAPPINGS
compatibility view (deprecated), 824
SYSCOLPERM
system view, 757
SYSCOLSTAT
system view, 758
SYSCOLSTATS
consolidated view, 811
SYSCOLUMN
compatibility view (deprecated), 825
SYSCOLUMNS
consolidated view, 811
SYSCONSTRAINT
system view, 759
SYSDEPENDENCY
system view, 760
SYSDOMAIN
system view, 761
SYSEVENT
system view, 761
SYSEVENTTYPE
system view, 763
SYSEXTERNLOGIN
system view, 763

SYSEXTERNLOGINS (see SYSEXTERNLOGIN system view)
SYSFILE
system view, 764
SYSFKCOL
compatibility view (deprecated), 825
SYSFKEY
system view, 765
SYSFOREIGNKEY
compatibility view (deprecated), 826
SYSFOREIGNKEYS
consolidated view, 812
SYSGROUP
system view, 766
SYSGROUPS
consolidated view, 812
SYSHISTORY
system view, 767
SYSIDX
system view, 768
SYSIDXCOL
system view, 770
SYSINDEX
compatibility view (deprecated), 826
SYSINDEXES
consolidated view, 813
SYSINFO
compatibility view (deprecated), 827
SYSIXCOL
compatibility view (deprecated), 828
SYSJAR
system view, 771
SYSJARCOMPONENT
system view, 771
SYSJAVACLASS
system view, 772
SYSLOGINMAP
system view, 773
SYSMVOPTION
system view, 774
SYSMVOPTIONNAME
system view, 774
SYSOBJECT
system view, 775
SYSOPTION
system view, 776
SYSOPTIONS
consolidated view, 814
SYSOPTSTAT
system view, 777
SYSPHYSIDX
system view, 777
SYSPROCAUTH
consolidated view, 814
SYSPROCEDURE
system view, 779
SYSPROCPARM
system view, 780
SYSPROCPARMS
consolidated view, 815
SYSPROCPERM
system view, 781
SYSPROCS
consolidated view, 814
SYSPROXYTAB
system view, 782
SYSPUBLICATION
system view, 783
SYSPUBLICATIONS
consolidated view, 815
SYSREMARK
system view, 784
SYSREMOTEOPTION
system view, 784
SYSREMOTEOPTION2
consolidated view, 816
SYSREMOTEOPTIONS
consolidated view, 816
SYSREMOTEOPTIONTYPE
system view, 785
SYSREMOTETYPE
system view, 785
SYSREMOTETYPES
consolidated view, 817
SYSREMOTEUSER
system view, 786
SYSREMOTEUSERS
consolidated view, 817
SYSSCHEDULE
system view, 788
SYSSERVER
system view, 789
SYSSOURCE
system view, 790
SYSSQLSERVERTYPE
system view, 790

SYSSSERVERS (see SYSSERVER system view)
SYSSUBSCRIPTION
  system view, 791
SYSSUBSCRIPTIONS
  consolidated view, 818
SYSSYNC
  system view, 792
SYSSYNC2
  consolidated view, 818
SYSSYNCPUBLICATIONDEFAULTS
  consolidated view, 819
SYSSYNCS
  consolidated view, 819
SYSSYNCSCRIPT
  system view, 793
SYSSYNCSCRIPTS
  consolidated view, 819
SYSSYNCSUBSCRIPTIONS
  consolidated view, 820
SYSSYNCUSERS
  consolidated view, 821
SYSTAB
  system view, 794
SYSTABAUTH
  consolidated view, 821
SYSTABCOL
  system view, 797
SYSTABLE
  compatibility view (deprecated), 828
SYSTABLEPERM
  system view, 799
system and catalog stored procedures
  about, 835
system calls
  from stored procedures, 946
  xp_cmdshell system procedure, 946
system catalog
  about, 726, 753
system extended procedures
  about, 951
system functions
  alphabetical list, 100
  compatibility, 101
system procedures
  about, 833
  alphabetical list , 835
  creating messages, 413
  extended list, 951
  overview, 834
  sa_flush_statistics, 868
  sa_set_soap_header, 924
  sp_addgroup, 962
  sp_addlogin, 962
  sp_addmessage, 962
  sp_addtype, 962
  sp_adduser, 962
  sp_changegroup, 962
  sp_dropgroup, 962
  sp_droplogin, 962
  sp_dropmessage, 962
  sp_droptype, 962
  sp_dropuser, 962
  sp_getmessage, 962
  sp_helptext, 962
  sp_password, 962
  Sybase Central, 834
  Transact-SQL, 962
  Transact-SQL list, 962
  viewing definitions, 834
SYSTEM statement
  Interactive SQL syntax, 691
system tables
  about, 726
  DUMMY, 726
  Java, 751
  RowGenerator, 751
system views
  about, 753
  SYSARTICLE, 754
  SYSARTICLECOL, 755
  SYSCAPABILITY, 755
  SYSCAPABILITYNAME, 756
  SYSCHECK, 757
  SYSCOLPERM, 757
  SYSCOLSTAT, 758
  SYSCONSTRAINT, 759
  SYSDEPENDENCY, 760
  SYSDOMAIN, 761
  SYSEVENT, 761
  SYSEVENTTYPE, 763
  SYSEXTERNLOGIN, 763
  SYSFILE , 764
  SYSFKEY, 765
  SYSGROUP, 766
  SYSHISTORY, 767
  SYSIDX, 768

SYSIDXCOL, 770
SYSJAR, 771
SYSJARCOMPONENT, 771
SYSJAVACLASS, 772
SYSLOGINMAP, 773
SYSMVOPTION, 774
SYSMVOPTIONNAME, 774
SYSOBJECT, 775
SYSOPTION, 776
SYSOPTSTAT, 777
SYSPHYSIDX, 777
SYSPROCEDURE, 779
SYSPROCPARM, 780
SYSPROCPERM, 781
SYSPROXYTAB, 782
SYSPUBLICATION, 783
SYSREMARK, 784
SYSREMOTEOPTION, 784
SYSREMOTEOPTIONTYPE, 785
SYSREMOTETYPE, 785
SYSREMOTEUSER, 786
SYSSCHEDULE, 788
SYSSERVER, 789
SYSSOURCE, 790
SYSSQLSERVERTYPE, 790
SYSSUBSCRIPTION, 791
SYSSYNC, 792
SYSSYNCSCRIPT, 793
SYSTAB, 794
SYSTABCOL, 797
SYSTABLEPERM, 799
SYSTRIGGER, 800
SYSTYPEMAP, 802
SYSUSER, 803
SYSUSERAUTHORITY, 804
SYSUSERMESSAGE, 804
SYSUSERTYPE, 805
SYSVIEW, 806
SYSWEBSERVICE, 807
SYSTRIGGER
system view, 800
SYSTRIGGERS
consolidated view, 822
SYSTYPEMAP
system view, 802
SYSUSER
system view, 803
SYSUSERAUTH
compatibility view (deprecated), 829
SYSUSERAUTHORITY
system view, 804
SYSUSERLIST
compatibility view (deprecated), 830
SYSUSERMESSAGE
system view, 804
SYSUSERMESSAGES (see SYSUSERMESSAGE system view)
SYSUSEROPTIONS
consolidated view, 822
SYSUSERPERM
compatibility view (deprecated), 830
SYSUSERPERMS
compatibility view (deprecated), 831
SYSUSERTYPE
system view, 805
SYSVIEW
system view, 806
SYSVIEWS
consolidated view, 823
SYSWEBSERVICE
system view, 807

T

table columns
listing in Interactive SQL, 494
table constraints
adding using the ALTER TABLE statement, 335
adding, deleting, or altering using the ALTER TABLE statement, 332
changing using ALTER TABLE statement, 336
CREATE TABLE statement, 456
table decryption
ALTER TABLE statement, 332
table encryption
ALTER TABLE statement, 332
table hints
FROM clause, 538
table indexes
listing in Interactive SQL, 494
table list
FROM clause, 536
table number
system views, 795
table pages
setting PCTFREE, 450, 483, 585

setting PCTFREE using the ALTER TABLE statement, 332
setting PCTFREE using the CREATE LOCAL TEMPORARY TABLE statement, 409
table-list
common element in SQL syntax, 296
table-name
common element in SQL syntax, 296
tables
ALTER TABLE statement, 332
altering using the ALTER TABLE statement, 332
bulk loading, 585
CREATE TABLE statement, 450
creating local temporary, 483
creating local temporary tables using the CREATE LOCAL TEMPORARY TABLE statement, 409
creating proxy tables using the CREATE EXISTING TABLE statement, 395
dropping using the DROP statement, 498
exporting data into files from, 604
importing data from files into, 568
inserting rows into, 573
locking, 593
renaming, 337
reorganizing, 628
truncating, 693
unloading with UNLOAD TABLE statement, 700
updating, 710
TAN function
SQL syntax, 265
tapes
creating database backups using the BACKUP statement, 346
technical support
newsgroups, xix
TempFreePercent event condition
about, 158
TempFreeSpace event condition
about, 158
temporary files
determining available space, 865
temporary options
SET OPTION statement, 664
setting in Interactive SQL, 667
temporary procedures
CREATE PROCEDURE reference, 416
temporary stored procedures
creating, 416
temporary tables
CREATE TABLE statement, 450
CREATE TABLE usage, 459
creating local temporary files using the CREATE LOCAL TEMPORARY TABLE statement, 409
declaring local, 483
Transact-SQL CREATE TABLE statement, 460
views disallowed on local, 471
TempSize event condition
about, 158
text
reading from the database, 620
TEXT data type
syntax, 53
text functions
about, 102
TEXTPTR function
SQL syntax, 265
textsize option
setting with Transact-SQL SET statement, 658
THEN
IF expressions, 17
three-valued logic
NULL value, 43
SQL syntax, 27
TIME data type
sending dates and times to the database, 67
syntax, 72
time data types
DATETIME, 72
overview, 67
SMALLDATETIME, 72
TIMESTAMP, 73
time functions
alphabetical list, 94
times
comparing, 69
conversion functions, 94
queries, 68
sending to the database, 67
TIMESTAMP
special value, 33
TIMESTAMP columns, 454
TIMESTAMP data type
sending dates and times to the database, 67
syntax, 73
TINYINT data type
syntax, 63

TO_CHAR function
SQL syntax, 266
TO_NCHAR function
SQL syntax, 267
TODAY function
SQL syntax, 268
TOP clause
SELECT statement, 648
TRACEBACK function
SQL syntax, 268
TRACED_PLAN function
SQL syntax, 269
tracing
ATTACH TRACING statement, 344
DETACH TRACING statement, 496
REFRESH TRACING LEVEL statement, 623
tracing data
saving using sa_save_trace_data system procedure, 912
tracing levels
setting the sa_set_tracing_level system procedure, 925
trademark information
retrieving, 960
Transact-SQL
alphabetical list of all statements, 294
ANSI equivalency, 236
bitwise operators, 13
BREAK statement syntax, 718
catalog procedures, 963
comparison operators, 11
constants, 19
CONTINUE statement syntax, 718
converting stored procedures, 278
CREATE FUNCTION statement, 404
CREATE MESSAGE SQL statement syntax, 413
CREATE PROCEDURE statement syntax, 425
CREATE SCHEMA statement syntax, 433
CREATE TABLE statement syntax, 460
CREATE TRIGGER statement syntax, 468
datetime compatibility, 68
DECLARE CURSOR statement syntax, 482
DECLARE section, 352
domains, 79
EXECUTE statement syntax, 517
GOTO statement syntax, 547
IF statement syntax, 565
local variables, 36
money data types, 64
outer join operators, 14
PRINT statement syntax, 613
quoted_identifier option, 19
RAISERROR statement syntax, 616
READTEXT statement syntax, 620
SET OPTION statement syntax, 658
SET statement syntax, 658
SQL expression compatibility, 19
statement indicators, 298
strings, 19
system functions, 101
system procedures, 962
time compatibility, 68
user-defined data types, 79
WHILE statement syntax, 718
WRITETEXT statement syntax, 722
Transact-SQL compatibility
global variables, 38
views, 831
Transact-SQL statements
BEGIN TRANSACTION syntax, 354
ROLLBACK TRANSACTION syntax, 644
SAVE TRANSACTION syntax, 646
Transact-SQL string-to-date/time conversions
about, 68
transaction isolation level option
setting with Transact-SQL SET statement, 658
transaction log
allocating space using ALTER DBSPACE, 305
backing up using the BACKUP statement, 346
determining available space, 865
TRUNCATE TABLE statement, 693
TRANSACTION LOG clause
CREATE DATABASE statement, 380
transaction log mirror
determining available space, 865
transaction management
BEGIN TRANSACTION SQL statement, 354
in Transact-SQL, 354
Transact-SQL, 367
transaction modes
chained, 354
unchained, 354
transactions
beginning user-defined using the BEGIN TRANSACTION statement, 354
committing using the COMMIT statement, 367

creating savepoints, 647
nesting user-defined transactions using the BEGIN TRANSACTION statement, 354
rolling back, 642, 644, 646
rolling back to savepoints, 643
TRANSACTSQL function
SQL syntax, 269
trapping
errors in embedded SQL, 717
trigger conditions
distinguishing trigger actions, 26
TRIGGER EVENT statement
SQL syntax, 692
triggering
events, 692
triggers
@@identity global variable, 41
altering using the ALTER TRIGGER statement, 341
creating in Transact-SQL, 468
creating using CREATE TRIGGER statement, 462
dropping using the DROP statement, 498
rolling back, 645
row-level, 464
statement-level, 464
TRUNCATE TABLE statement, 694
TRIM function
SQL syntax, 270
troubleshooting
locks, 882
logging operations, 917
newsgroups, xix
non-standard disk drives, 303
TRUE conditions
IS TRUE search condition, 26
three-valued logic, 27
TRUNCATE function
SQL syntax, 270
TRUNCATE TABLE statement
SQL syntax, 693
truncating
tables, 693
TRUNCNUM function
SQL syntax, 270
TSQL (see Transact-SQL)
two-phase commit
preparing for, 612
TYPE clause
CREATE SYNCHRONIZATION USER, 448
type conversion
about, 80
types of data (see data types)

U

UCASE function
SQL syntax, 271
UNBOUNDED keyword
PRECEDING clause of WINDOW clause, 720
undoing
changes by rolling back transactions, 642
Unicode data
storage, 48
Unicode data types
about, 48
UNICODE function
SQL syntax, 272
UNION statement
SQL syntax, 695
unions
multiple select statements, 695
unique
constraint in CREATE TABLE statement, 456
unique indexes
about, 405
UNIQUEIDENTIFIER data type
syntax, 75
UNIQUEIDENTIFIERSTR data type
syntax, 53
UNISTR function
SQL syntax, 272
universally unique identifiers
SQL syntax for NEWID function, 204
Unix
compressing strings, 121
decompressing strings, 148
UNKNOWN conditions
IS UNKNOWN search condition, 26
UNLOAD statement
SQL syntax, 698
UNLOAD TABLE statement
SQL syntax, 700
unloading
materialized views, 700
result sets, 698
tables, 700

unloading data
multibyte character sets, 701
unzip utility
DECOMPRESS function, 148
updatable views
about, 575
UPDATE (positioned) statement
SQL syntax, 708
UPDATE clause
CREATE TRIGGER [Transact-SQL], 468
CREATE TRIGGER [Transact-SQL] statement, 468
CREATE TRIGGER statement, 462
update column permission
SYSCOLPERM system view, 757
UPDATE statement
SQL Remote SQL syntax, 710
SQL syntax, 703
UPDATE statement [SQL Remote]
SQL syntax, 710
updates
based on joins, 705
joins, 711
updating
columns without logging, 722
publications and subscriptions, 706
rows, 703
tables and columns, 710
UPDATING condition
triggers, 26
UPDLOCK table hint
FROM clause, 538
upgrading databases
ALTER DATABASE statement, 301
UPPER function
SQL syntax, 273
uppercase characters
UPPER function, 273
uppercase strings
UCASE function, 271
UPPER function, 273
USER
special value, 34
user estimates
about, 28
user IDs
restrictions, 549
revoking, 636
system views, 795
views, 829
user-defined data types
about, 78
CREATE DOMAIN statement, 386
dropping using the DROP statement, 498
Transact-SQL, 79
user-defined functions
alphabetical list, 96
CREATE FUNCTION statement, 399
exiting from, 634
Java, 96
returning values from, 634
user-supplied selectivity estimates
about, 28
userid
common element in SQL syntax, 296
users
ALTER SYNCHRONIZATION USER statement, 330
CREATE SYNCHRONIZATION USER statement, 448
DROP SYNCHRONIZATION USER statement, 511
dropping, 636
setting, 671
using the SQL statement reference
about, 295
using unambiguous dates and times
about, 70
ust files
creating, 249
UTC TIMESTAMP
special value, 35
UUIDs
SQL syntax for NEWID function, 204
SQL syntax for STRTOUUID function, 261
SQL syntax for UUIDTOSTR function, 274
UNIQUEIDENTIFIER data type, 75
UUIDTOSTR function
SQL syntax, 274

V

VALIDATE CHECKSUM statement
SQL syntax, 713
VALIDATE DATABASE statement
SQL syntax, 713

VALIDATE INDEX statement
SQL syntax, 713
VALIDATE MATERIALIZED VIEW statement
SQL syntax, 713
VALIDATE statement
SQL syntax, 713
VALIDATE TABLE statement
SQL syntax, 713
validating
checksums, 713
databases, 934
indexes using VALIDATE statement, 713
tables using VALIDATE TABLE statement, 713
VALIDATE statement, 713
validation
VALIDATE permission, 548
values
returning from procedures, 634
VAR_POP function
SQL syntax, 275
VAR_SAMP function
SQL syntax, 276
VARBINARY data type
syntax, 76
VARBIT data type
syntax, 65
VARCHAR data type
byte-length semantics, 53
character-length semantics, 53
syntax, 53
using DESCRIBE on a VARCHAR column, 53
VAREXISTS function
SQL syntax, 278
variable result sets
from procedures, 416, 491, 610
variable-name
common element in SQL syntax, 296
variables
connection-level variables, 37
creating SQL, 469
declaring SQL, 477
dropping SQL variables using the DROP VARIABLE statement, 512
getting from within a descriptor area, 544
global variables, 38
local variables, 36
setting values, 656
SQL syntax, 36
using in view definitions, 471
VARIANCE function
SQL syntax, 278
verifying
passwords, 936
version number
retrieving, 960
view dependencies
unloading/reloading databases, 906
viewing
Interactive SQL procedure profiling data, 904
views
altering using the ALTER VIEW statement, 342
compatibility views, 824
consolidated views, 809
CREATE MATERIALIZED VIEW statement, 411
CREATE VIEW statement, 471
DROP statement, 498
indexes, 407
parameterized views, 471
sa_recompile_views system procedure, 906
system views, 754
Transact-SQL compatibility, 831
updatable, 575
VIM message type
DROP REMOTE MESSAGE TYPE statement, 504
SQL Remote ALTER REMOTE MESSAGE TYPE statement, 319
SQL Remote CREATE REMOTE MESSAGE TYPE statement, 431
VM
START JAVA statement, 677
STOP JAVA statement, 685

W

WAITFOR statement
SQL syntax, 715
Watcom-SQL
DECLARE statement, 477
Watcom-SQL statements
rewriting to Transact-SQL, 269
WATCOMSQL function
SQL syntax, 278
web servers
altering services using the ALTER SERVICE statement, 323

creating, 438
dropping using the DROP SERVICE statement, 506
web services
adding comments using the COMMENT statement, 365
sa_set_http_option system procedure, 922
sa_set_soap_header system procedure, 924
system view, 807
WEEKS function
SQL syntax, 279
WHEN
CASE expression, 17
WHENEVER statement
embedded SQL syntax, 717
WHERE clause
search conditions, 20
SELECT statement, 651
WHILE statement
SQL syntax, 595
Transact-SQL syntax, 718
wide inserts
about, 515
wildcards
LIKE search condition, 23
PATINDEX function, 212
WINDOW clause
SELECT statement, 651
SQL syntax, 719
window functions
AVG function, 107
COUNT function, 129
COVAR_POP function, 131
CUME_DIST function, 135
DENSE_RANK function, 151
MAX function, 198
MIN function, 199
PERCENT_RANK function, 213
RANK function, 221
REGR_AVGX function, 222
REGR_AVGY function, 223
REGR_COUNT function, 224
REGR_INTERCEPT function, 225
REGR_R2 function, 227
REGR_SLOPE function, 228
REGR_SXX function, 229
REGR_SXY function, 230
ROW_NUMBER function, 240
STDDEV function, 257
STDDEV_POP function, 257
STDDEV_SAMP function, 258
SUM function, 264
VAR_POP function, 275
VAR_SAMP function, 276
window-name
common element in SQL syntax, 297
window-spec
syntax in window functions, 719
windows (OLAP)
WINDOW clause, 719
WITH CHECKPOINT option
LOAD TABLE statement, 589
WITH clause
SELECT statement, 648
WITH GRANT OPTION
SQL syntax, 548
WITH HOLD clause
OPEN SQL statement, 601
WITH RECURSIVE clause
SELECT statement, 648
WITH SCRIPTED UPLOAD clause
CREATE PUBLICATION statement, 427
words
reserved, 4
WRITETEXT statement
Transact-SQL syntax, 722

X

XLOCK table hint
FROM clause, 538
XML
XML data type, 54
XMLAGG function, 280
XMLCONCAT function, 281
XMLELEMENT function, 282
XMLFOREST function, 284
XMLGEN function, 285
XML data type
syntax, 54
XMLAGG function
SQL syntax, 280
XMLCONCAT function
SQL syntax, 281
XMLELEMENT function
SQL syntax, 282

XMLFOREST function
SQL syntax, 284
XMLGEN function
SQL syntax, 285
xp_cmdshell system procedure
syntax, 946
xp_msver system procedure
syntax, 960
xp_read_file system procedure
syntax, 947
xp_scanf system procedure
syntax, 947
xp_sendmail system procedure
syntax, 953
xp_sprintf system procedure
syntax, 948
xp_startmail system procedure
syntax, 951
xp_startsmtp system procedure
enabling in McAfee® VirusScan, 953
possible conflicts with virus scanner settings, 953
syntax, 952
xp_stopmail system procedure
syntax, 957
xp_stopsmtp system procedure
syntax, 957
xp_write_file system procedure
syntax, 949

Y

YEAR function
SQL syntax, 286
YEARS function
SQL syntax, 286
YMD function
SQL syntax, 288

Z

zip utility
COMPRESS function, 121

