Adaptive Server® Anywhere SQL Reference

Part number: DC38129-01-0902-01
Last modified: October 2004

Contents

About This Manual

Subject	This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables 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 Adaptive Server Anywhere. It includes material of particular interest to users of MobiLink and SQL Remote. It is to be used in conjunction with other manuals in the documentation set.

SQL Anywhere Studio 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 Studio documentation

The SQL Anywhere Studio documentation is available in a variety of forms: in an online form that combines all books in one large help file; as separate PDF files for each book; and as printed books that you can purchase. The documentation consists of the following books:

♦ **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.

♦ **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.

♦ **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases and database servers.

♦ **Adaptive Server Anywhere SQL User's Guide** 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.

♦ **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.

♦ **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.

♦ **Adaptive Server Anywhere SNMP Extension Agent User's Guide** This book describes how to configure the Adaptive Server Anywhere SNMP Extension Agent for use with SNMP management applications to manage Adaptive Server Anywhere databases.

♦ **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.

- **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.
- **MobiLink Administration Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.
- **MobiLink Clients** This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.
- **MobiLink Server-Initiated Synchronization User's Guide** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.
- **MobiLink Tutorials** This book provides several tutorials that walk you through how to set up and run MobiLink applications.
- **QAnywhere User's Guide** This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.
- **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.
- **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.
- **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.
- **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.

- **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

In addition to this documentation set, PowerDesigner and InfoMaker include their own online documentation.

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere Studio documentation, including both 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 9 ➤ 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.

- **PDF books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

  The PDF books are accessible from the online books, or from the Windows Start menu.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store at *http://eshop.sybase.com/eshop/documentation*.

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 upper case, 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.

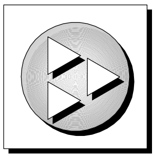
  [**QUOTES** { **ON** | **OFF** }]

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

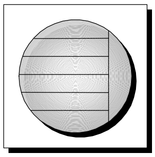
Graphic icons

The following icons are used in this documentation.

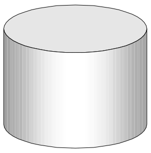
♦ A client application.



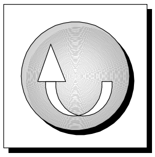
♦ A database server, such as Sybase Adaptive Server Anywhere.



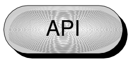
♦ A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



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



♦ A programming interface.



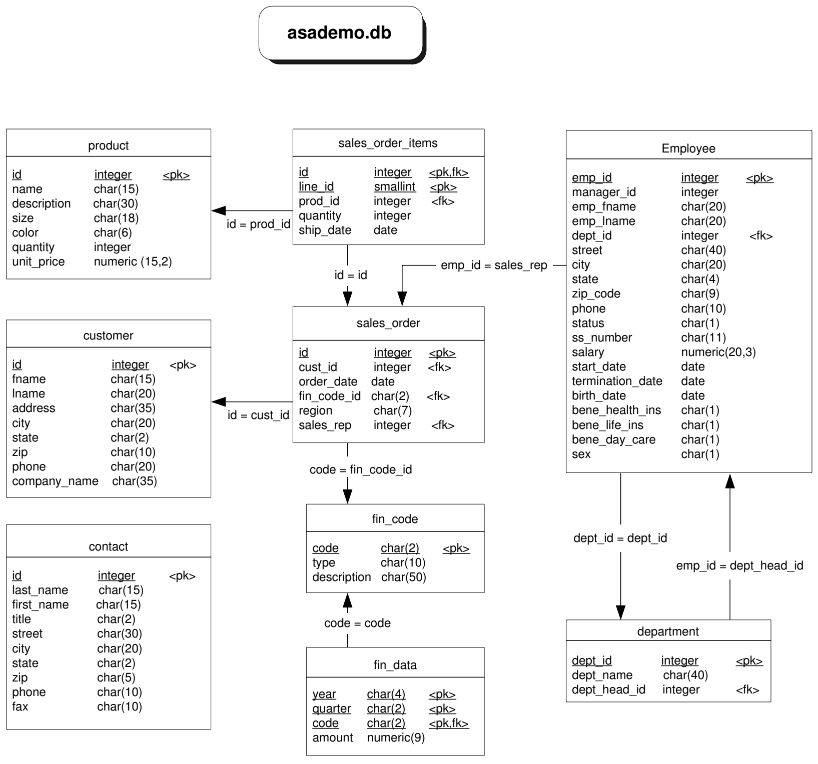
The Adaptive Server Anywhere sample database

Many of the examples throughout the documentation use the Adaptive Server Anywhere sample database.

The sample database is held in a file named *asademo.db*, and is located in your SQL Anywhere directory.

The sample database represents a small company. It contains internal information about the company (employees, departments, and finances) as well as product information and sales information (sales orders, customers, and contacts). All information in the database is fictional.

The following figure shows the tables in the sample database and how they relate to each other.



Finding out more and providing feedback

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

You can provide feedback on this documentation and on the software through newsgroups set up to discuss SQL Anywhere technologies. These newsgroups can be found 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 insure its operation and availability.

iAnywhere Solutions 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.

You can e-mail comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not undertake to reply to e-mails at that address, you can be sure we will read your suggestions with interest.

PART I

SQL

This part describes the Adaptive Server Anywhere SQL language, including data types, functions and statements.

CHAPTER 1

SQL Language Elements

About this chapter

This chapter describes the elements and conventions of the SQL language.

Contents

Keywords

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

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

```
SELECT *
FROM employee
```

The following statements are equivalent to the one above:

```
Select *
From employee
select * from employee
sELECT * FRoM employee
```

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 upper case, lower case, 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 database library function **SQL_needs_quotes** 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.

The SQL keywords in Adaptive Server Anywhere are as follows:

Reserved word	Reserved word	Reserved word	Reserved word
add	**all**	**alter**	**and**
any	**as**	**asc**	**backup**

Reserved word	Reserved word	Reserved word	Reserved word
begin	**between**	**bigint**	**binary**
bit	**bottom**	**break**	**by**
call	**capability**	**cascade**	**case**
cast	**char**	**char_convert**	**character**
check	**checkpoint**	**close**	**comment**
commit	**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**
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	**key**	**lateral**	**left**
like	**lock**	**login**	**long**
match	**membership**	**message**	**mode**

Reserved word	Reserved word	Reserved word	Reserved word
modify	**natural**	**new**	**no**
noholdlock	**not**	**notify**	**null**
numeric	**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**	**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**
varchar	**variable**	**varying**	**view**
wait	**waitfor**	**when**	**where**
while	**window**	**with**	**with_cube**
with_lparen	**with_rollup**	**within**	**work**
writetext			

Identifiers

Function

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

Description

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.

You can represent an apostrophe (single quote) inside an identifier by following it with another apostrophe.

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]" [*ASA Database Administration Guide*, page 685].

Examples

The following are all valid identifiers.

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

Strings

Strings are of the following types:

- literal strings
- expressions with CHAR or VARCHAR data types.

An expression with a CHAR data type may be a built-in or user-defined function, or one of the many other kinds of expressions available.

☞ For more information on expressions, see "Expressions" on page 16.

A literal string is any sequence of characters enclosed in apostrophes ('single quotes'). A SQL variable of character data type can hold a string. The following is a simple example of a literal string:

```
'This is a string.'
```

Special characters in strings

You represent special character in strings by escape sequences, as follows:

- To represent an apostrophe inside a string, use two apostrophes in a row. For example,

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

- To represent a new line character, use a backslash followed by n (\n). For example,

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

- To represent a backslash character, use two backslashes in a row (\\). For example,

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

- Hexadecimal escape sequences can be used for any character, printable or not. A hexadecimal escape sequence is a backslash followed by an x followed by two hexadecimal digits (for example, \x6d represents the letter m). For example,

  ```
  '\x00\x01\x02\x03'
  ```

See also

- For information about the handling of strings in dynamically constructed SQL statements, see "EXECUTE IMMEDIATE statement [SP]" on page 475.

Standards and compatibility

For compatibility with Adaptive Server Enterprise, you can set the QUOTED_IDENTIFIER database option to OFF. With this setting, you can also use double quotes to mark the beginning and end of strings. The option is set to ON by default.

☞ For information about the QUOTED_IDENTIFIER option, see "QUOTED_IDENTIFIER option [compatibility]" [*ASA Database Administration Guide,* page 685].

Operators

This section describes arithmetic, string, and bit-wise operators. For information on comparison operators, see the section "Search conditions" on page 23.

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.

Standards and compatibility

- **Trailing blanks** Any trailing blanks in character data are ignored for comparison purposes by Adaptive Server Enterprise. The behavior of Adaptive Server Anywhere when comparing strings is controlled the `-b` command-line switch that is set when creating the database.

☞ For more information about blank padding, see “Initialization utility options” [*ASA Database Administration Guide*, page 532].

- **Case sensitivity** By default, Adaptive Server Anywhere databases are created as case insensitive, while Adaptive Server Enterprise databases are created as case sensitive. Comparisons are carried out with the same attention to case as the database they are operating on. You can control the case sensitivity of Adaptive Server Anywhere databases with the -c command line switch when you create the database.

  ☞ For more information about case sensitivity for string comparisons, see “Initialization utility options” [*ASA Database Administration Guide*, page 532].

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 30.

Standards and compatibility

- The logical operators are compatible between Adaptive Server Anywhere and Adaptive Server Enterprise.

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 Adaptive Server 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/92** The || operator is the SQL/92 string concatenation operator.
- **Sybase** The + operator is supported by Adaptive Server Enterprise.

The || concatenation operator is not supported by Adaptive Server Enterprise.

Bitwise operators

The following operators can be used on integer data types, in both Adaptive Server Anywhere and Adaptive Server Enterprise.

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 Transact-SQL outer join operators *= and =* are supported in Adaptive Server Anywhere, in addition to the SQL/92 join syntax that uses a table expression in the FROM clause.

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 rather than relying on an identical operator precedence between Adaptive Server Enterprise and Adaptive Server Anywhere.

Expressions

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

Usage

Anywhere.

Authorization	Must be connected to the database.
Side effects	None.

Description

Expressions are formed from several different kinds of elements. These are discussed in the following sections.

☞ For information on functions, see "SQL Functions" on page 91. For information on variables, see "Variables" on page 38.

See also

- ♦ "Constants in expressions" on page 17
- ♦ "Special values" on page 33
- ♦ "Column names in expressions" on page 17
- ♦ "SQL Functions" on page 91
- ♦ "Subqueries in expressions" on page 17
- ♦ "Search conditions" on page 23
- ♦ "SQL Data Types" on page 53
- ♦ "Variables" on page 38
- ♦ "CASE expressions" on page 18

Standards and compatibility

- ♦ The IF condition is not supported in Adaptive Server Enterprise.
- ♦ 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 491.) 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:

```
employee.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 24.

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 49, and “Search conditions” on page 23.

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 518.

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 "DBA".Product
```

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 "DBA".Product
```

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 320.

Compatibility of expressions

The following tables describe the compatibility of expressions and constants between Adaptive Server Enterprise and Adaptive Server Anywhere. These tables are a guide only, and a marking of Both may not mean that the expression performs in an identical manner for all purposes under all circumstances. For detailed descriptions, you should refer to the Adaptive Server Enterprise documentation and the Adaptive Server Anywhere documentation on the individual expression.

In the following table, **expr** represents an expression, and **op** represents an operator.

Expression	Supported by
constant	Both
column name	Both
variable name	Both
function (expr)	Both
- expr	Both
expr op expr	Both
(expr)	Both
(subquery)	Both
if-expression	Adaptive Server Anywhere only

Constant	Supported by
integer	Both
number	Both
‘string’	Both
special-constant	Both
host-variable	Adaptive Server Anywhere

Default interpretation of delimited strings

By default, Adaptive Server Enterprise and Adaptive Server Anywhere give different meanings to delimited strings: that is, strings enclosed in apostrophes (single quotes) and in quotation marks (double quotes).

Adaptive Server Anywhere employs the SQL/92 convention, that strings enclosed in apostrophes are constant expressions, and strings enclosed in quotation marks (double quotes) are delimited identifiers (names for database objects). Adaptive Server Enterprise employs the convention that strings enclosed in quotation marks are constants, while delimited identifiers are not allowed by default and are treated as strings.

The quoted_identifier option

Both Adaptive Server Enterprise and Adaptive Server Anywhere provide a **quoted_identifier** option that allows the interpretation of delimited strings to be changed. By default, the **quoted_identifier** option is set to OFF in Adaptive Server Enterprise, and to ON in Adaptive Server Anywhere.

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

While the Transact-SQL SET statement is not supported for most Adaptive Server Enterprise connection options, it is supported for the **quoted_identifier** option.

The following statement in either Adaptive Server Anywhere or Adaptive Server Enterprise changes the setting of the **quoted_identifier** option to ON:

```
SET quoted_identifier ON
```

With the **quoted_identifier** option set to ON, Adaptive Server Enterprise allows table, view, and column names to be delimited by quotes. Other object names cannot be delimited in Adaptive Server Enterprise.

The following statement in Adaptive Server Anywhere or Adaptive Server Enterprise 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/92 or the default Transact-SQL convention in both Adaptive Server Enterprise and Adaptive Server 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 Adaptive Server 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 (the default Adaptive Server Enterprise setting), then the following statements are valid for both DBMSs.

```
SELECT *
FROM employee
WHERE emp_lname = "Chin"
```

Search conditions

Function

To specify a search condition 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* , *value-expr2* [,*value-expr3*] . . . })
| **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* **)**

Usage

Anywhere.

Permissions

Must be connected to the database.

Side effects

None.

Description

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 the section "Three-valued logic" on page 30.

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 49.

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 24.

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

See also

- ♦ "Expressions" on page 16

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 one column) returns one row, then the value of that row is compared to the expression. If a subquery returns no rows, its value is NULL.

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

ALL or ANY conditions

The syntax for ANY conditions is

expression compare **ANY (** *subquery* **)**

where *compare* is a comparison operator.

For example, an ANY condition with an equality operator,

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

is TRUE if *expression* is equal to any of the values in the result of the subquery, and FALSE if the expression is not NULL and does not equal any of the columns of the subquery. The ANY condition is UNKNOWN if *expression* is the NULL value, unless the result of the subquery has no rows, in which case the condition is always FALSE.

The keyword **SOME** can be used instead of **ANY**.

The syntax for ALL conditions is

expression compare **ALL** (*subquery*) where *compare* is a comparison operator.

Standards and compatibility

- ANY and ALL subqueries are compatible between Adaptive Server Enterprise and Adaptive Server Anywhere. Only Adaptive Server Anywhere supports SOME as a synonym for ANY.

BETWEEN conditions

The syntax for BETWEEN conditions is as follows:

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

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

The BETWEEN conditions is equivalent to a combination of two inequalities:

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

Standards and compatibility

- The BETWEEN condition is compatible between Adaptive Server Anywhere and Adaptive Server Enterprise.

LIKE conditions

The syntax for LIKE conditions is as follows:

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

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

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

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

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

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-expr* 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 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 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" [*ASA Database Administration Guide*, page 319].

The lower end of the range must precede the higher end of the range. For example, a LIKE 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 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 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 condition finds the strings *bough*, *rough*, and *tough*:

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

The bracket [*a-mpqs-z*] is interpreted as "exactly one character that is either in the range *a* to *m* inclusive, or is *p*, or is *q*, or is in the range *s* to *z* inclusive".

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 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-mpqs-z]* is interpreted as "exactly one character that is not in the range *a* to *m* inclusive, is not *p*, is not *q*, and is not in the range *s* to *z* inclusive".

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, Adaptive Server 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.

Standards and compatibility

- The ESCAPE clause is supported by Adaptive Server Anywhere only.

IN conditions

The syntax for IN conditions is as follows:

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

An IN 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 identical to the search condition *expression* = **ANY** (*values*). The search condition *expression* **NOT IN** (*values*) is identical to the search condition *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 datatype.

Standards and compatibility

- IN conditions are compatible between Adaptive Server Enterprise and Adaptive Server Anywhere.

EXISTS conditions

The syntax for EXISTS conditions is as follows:

EXISTS(*subquery* **)**

The EXISTS 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 condition cannot be UNKNOWN.

Standards and compatibility

- The EXISTS condition is compatible between Adaptive Server Enterprise and Adaptive Server Anywhere.

IS NULL conditions

The syntax for IS NULL conditions is as follows:

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

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

Standards and compatibility

- The IS NULL condition is compatible between Adaptive Server Enterprise and Adaptive Server Anywhere.

Truth value conditions

The syntax for truth-value conditions is as follows:

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

Without the NOT keyword, the 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 condition, but leaves UNKNOWN unchanged.

Standards and compatibility

- Vendor extension. Adaptive Server Enterprise does not support truth-valued conditions.

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
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

Explicit selectivity estimates

Adaptive Server Anywhere uses statistical information to determine the most efficient strategy for executing each statement. Adaptive Server Anywhere automatically gathers and updates these statistics. These statistics are stored permanently in the database in the system table SYSCOLSTAT. 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 491.

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]" [*ASA Database Administration Guide*, page 700].

☞ For more information about statistics, see "Optimizer estimates and histograms" [*ASA SQL User's Guide*, page 397].

Examples

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

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

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

```
SELECT *
    FROM customer c, sales_order o
WHERE (c.id = o.cust_id, 0.5)
```

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

Standards and compatibility

- Adaptive Server Enterprise does not support explicit estimates.

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

Function	CURRENT DATABASE returns the name of the current database.
Data type	STRING
See also	♦ “Expressions” on page 16

CURRENT DATE special value

Function	CURRENT DATE returns the current year, month, and day.
Data type	DATE
See also	♦ “Expressions” on page 16 ♦ “TIME data type [Date and Time]” on page 72

CURRENT PUBLISHER special value

Function	CURRENT PUBLISHER returns a string that contains the publisher user ID of the database for SQL Remote replications.
Data type	STRING CURRENT PUBLISHER can be used as a default value in columns with character data types.
See also	♦ “Expressions” on page 16 ♦ “SQL Remote Design for Adaptive Server Anywhere” [*SQL Remote User’s Guide*, page 91]

CURRENT TIME special value

Function	The current hour, minute, second and fraction of a second.
Data type	TIME
Description	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 16 ♦ “TIME data type [Date and Time]” on page 72

CURRENT TIMESTAMP special value

Function	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. The information CURRENT TIMESTAMP returns is equivalent to the information returned by the GETDATE and NOW functions. CURRENT_TIMESTAMP is equivalent to CURRENT TIMESTAMP.
Data type	TIMESTAMP
See also	♦ “CURRENT TIME special value” on page 33 ♦ “Expressions” on page 16 ♦ “TIMESTAMP data type [Date and Time]” on page 73 ♦ “GETDATE function [Date and time]” on page 157 ♦ “NOW function [Date and time]” on page 189

CURRENT USER special value

Function	CURRENT USER returns a string that contains the user ID of the current connection.
Data type	STRING CURRENT USER can be used as a default value in columns with character data types.
Description	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 16

CURRENT UTC TIMESTAMP special value

Function	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 [Date and Time]” on page 73
- ♦ “UTC TIMESTAMP special value” on page 36
- ♦ “CURRENT TIMESTAMP special value” on page 34
- ♦ “TRUNCATE_TIMESTAMP_VALUES option [database]” [*ASA Database Administration Guide*, page 697]

LAST USER special value

Function

The name of the user who last modified the row.

Data type

String.

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

Description

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 34
- ♦ “CURRENT TIMESTAMP special value” on page 34
- ♦ “CREATE TABLE statement” on page 407

SQLCODE special value

Function

Current SQLCODE value.

Data type

String.

Description

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 16
- ♦ *ASA Error Messages*.

SQLSTATE special value

Function

Current SQLSTATE value

Data type

STRING

Description

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 16

- *ASA Error Messages*

TIMESTAMP special value

Function	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
Description	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 Adaptive Server Anywhere based on the DEFAULT_TIMESTAMP_INCREMENT option. This is useful for maintaining compatibility with other database software which records less precise timestamp values.

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

See also

- "TIMESTAMP data type [Date and Time]" on page 73
- "CURRENT UTC TIMESTAMP special value" on page 34
- "DEFAULT_TIMESTAMP_INCREMENT option [database]" [*ASA Database Administration Guide*, page 650]
- "TRUNCATE_TIMESTAMP_VALUES option [database]" [*ASA Database Administration Guide*, page 697]

USER special value

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

See also

- "Expressions" on page 16
- "CURRENT USER special value" on page 34
- "LAST USER special value" on page 35

UTC TIMESTAMP special value

Function

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

Data type

TIMESTAMP

See also

- “TIMESTAMP data type [Date and Time]” on page 73
- “CURRENT UTC TIMESTAMP special value” on page 34
- “TIMESTAMP special value” on page 36
- “DEFAULT_TIMESTAMP_INCREMENT option [database]” [*ASA Database Administration Guide*, page 650]
- “TRUNCATE_TIMESTAMP_VALUES option [database]” [*ASA Database Administration Guide*, page 697]

Variables

Adaptive Server 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 [Date and Time]" on page 73
- ♦ "CREATE VARIABLE statement" on page 427

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). 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 pane.

- 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 window.

Standards and compatibility

- **Names** Adaptive Server Enterprise and Adaptive Server Anywhere both support local variables. In Adaptive Server Enterprise, all variables must be prefixed with an @ sign. In Adaptive Server Anywhere, the @ prefix is optional. To write compatible SQL, prefix all of your variables with @.

- **Scope** The scope of local variables is different in Adaptive Server Anywhere and Adaptive Server Enterprise. Adaptive Server Anywhere supports the use of the DECLARE statement to declare local variables within a batch. However, if the DECLARE is executed within a compound statement, the scope is limited to the compound statement.

- **Declaration** Only one variable can be declared for each DECLARE statement in Adaptive Server Anywhere. In Adaptive Server Enterprise, more than one variable can be declared in a single statement.

For more information on batches and local variable scope, see "Variables in Transact-SQL procedures" [*ASA SQL User's Guide*, page 502].

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
```

Standards and compatibility

- ♦ Adaptive Server Enterprise does not support connection-level variables.

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 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 Adaptive Server Anywhere

Variable name	Meaning
@@dbts	A value of type TIMESTAMP representing the last generated value used for all columns defined with DEFAULT TIMESTAMP.
@@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 SQL statement resets @@error, 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. @@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. This feature is the same as @@sqlstatus, except that it returns different values. It is for Microsoft SQL Server compatibility.
@@identity	Last value inserted into any IDENTITY or DEFAULT AUTOINCREMENT column by an INSERT or SELECT INTO statement. For a description, see "@@identity global variable" on page 46.
@@isolation	Current isolation level. @@isolation takes the value of the active level.

Variable name	Meaning
@@procid	Stored procedure ID of the currently executing procedure.
@@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. Unlike in Adaptive Server Enterprise, @@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.
@@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.
@@version	Version number of the current version of Adaptive Server Anywhere.

Standards and compatibility

The following list includes all Adaptive Server Enterprise global variables supported in Adaptive Server Anywhere. Adaptive Server Enterprise global variables not supported by Adaptive Server Anywhere are not included in the list. In contrast to the above table, this list includes all global variables that return a value, including those for which the value is fixed at NULL, 1, -1, or 0, and may not be meaningful.

Global variable	Returns
@@char_convert	Returns 0.

Global variable	Returns
@@client_csname	In Adaptive Server Enterprise, the client's character set name. Set to NULL if client character set has never been initialized; otherwise, it contains the name of the most recently used character set. Returns NULL in Adaptive Server Anywhere.
@@client_csid	In Adaptive Server Enterprise, the client's character set ID. Set to –1 if client character set has never been initialized; otherwise, it contains the most recently used client character set ID from syscharsets. Returns –1 in Adaptive Server Anywhere.
@@connections	The number of logins since the server was last started
@@cpu_busy	In Adaptive Server Enterprise, the amount of time, in ticks, that the CPU has spent doing Adaptive Server Enterprise work since the last time Adaptive Server Enterprise was started. In Adaptive Server Anywhere, returns 0.
@@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.
@@identity	Last value inserted into an IDENTITY column by an INSERT or SELECT INTO statement. For a description, see "@@identity global variable" on page 46.
@@idle	In Adaptive Server Enterprise, the amount of time, in ticks, that Adaptive Server Enterprise has been idle since it was last started. In Adaptive Server Anywhere, returns 0.
@@io_busy	In Adaptive Server Enterprise, the amount of time, in ticks, that Adaptive Server Enterprise has spent doing input and output operations since it was last started. In Adaptive Server Anywhere, returns 0.

Global variable	Returns
@@isolation	Current isolation level of the connection. In Adaptive Server Enterprise, @@isolation takes the value of the active level
@@langid	Returns a unique language ID for the language in use by the current connection.
@@language	Returns the name of the language in use by the connection.
@@maxcharlen	In Adaptive Server Enterprise, maximum length, in bytes, of a character in Adaptive Server Enterprise's default character set. In Adaptive Server Anywhere, returns 1.
@@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). For Adaptive Server Enterprise, the maximum number of connections to the server.
@@ncharsize	In Adaptive Server Enterprise, average length, in bytes, of a national character. In Adaptive Server Anywhere, returns 1.
@@nestlevel	In Adaptive Server Enterprise, nesting level of current execution (initially 0). Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. In Adaptive Server Anywhere, returns –1.
@@pack_received	In Adaptive Server Enterprise, number of input packets read by Adaptive Server Enterprise since it was last started. In Adaptive Server Anywhere, returns 0.
@@pack_sent	In Adaptive Server Enterprise, number of output packets written by Adaptive Server Enterprise since it was last started. In Adaptive Server Anywhere, returns 0.

Global variable	Returns
@@packet_errors	In Adaptive Server Enterprise, number of errors that have occurred while Adaptive Server Enterprise was sending and receiving packets. In Adaptive Server Anywhere, returns 0.
@@procid	Stored procedure ID of the currently executing procedure.
@@rowcount	Number of rows affected by the last command. In Adaptive Server Enterprise @@rowcount is set to zero by any command which does not return rows, such as an *IF* statement; in Adaptive Server Anywhere, such statements to not reset @@rowcount. With cursors, @@rowcount represents the cumulative number of rows returned from the cursor result set to the client, up to the last fetch request.
@@servername	Name of the local Adaptive Server Enterprise or Adaptive Server Anywhere server.
@@spid	In Adaptive Server Enterprise, server process ID number of the current process. In Adaptive Server Anywhere, 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	In Adaptive Server Enterprise, change in free space required to activate a threshold. In Adaptive Server Anywhere, returns 0.

Global variable	Returns
@@timeticks	In Adaptive Server Enterprise, number of microseconds per tick. The amount of time per tick is machine-dependent. In Adaptive Server Anywhere, returns 0.
@@total_errors	In Adaptive Server Enterprise, number of errors that have occurred while Adaptive Server Enterprise was reading or writing. In Adaptive Server Anywhere, returns 0.
@@total_read	In Adaptive Server Enterprise, number of disk reads by Adaptive Server Enterprise since it was last started. In Adaptive Server Anywhere, returns 0.
@@total_write	In Adaptive Server Enterprise, number of disk writes by Adaptive Server Enterprise since it was last started. In Adaptive Server Anywhere, returns 0.
@@tranchained	Current transaction mode of the Transact-SQL program. @@tranchained returns 0 for unchained or 1 for chained.
@@trancount	Nesting level of transactions. Each BEGIN TRANSACTION in a batch increments the transaction count.
@@transtate	In Adaptive Server Enterprise, current state of a transaction after a statement executes. In Adaptive Server Anywhere, returns –1.
@@version	Information on the current version of Adaptive Server Enterprise or Adaptive Server 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 Adaptive Server Anywhere.

- **– (Double hyphen)** The database server ignores any remaining characters on the line. This is the SQL/92 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.

Standards and compatibility

- The double-hyphen and the slash-asterisk comment styles are compatible with Adaptive Server Enterprise.

Examples

- The following example illustrates the use of double-dash 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 SalesEmployee AS
SELECT emp_id, emp_lname, emp_fname
FROM "DBA".employee
WHERE dept_id = 200
```

NULL value

Function	To specify a value that is unknown or not applicable.
Syntax	**NULL**
Usage	Anywhere.
Permissions	Must be connected to the database.
Side effects	None.
See also	♦ "Expressions" on page 16 ♦ "Search conditions" on page 23

Description

The NULL value is a special value which is different from any valid value for any data type. However, the NULL value is a legal value in any data type. The NULL value 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 23 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

Condition	Truth value	Selected?
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 23 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
```

Standards and compatibility

- **SQL/92** Entry-level feature.
- **Sybase** In some contexts, Adaptive Server Enterprise treats NULL as a value, whereas Adaptive Server 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 Adaptive Server 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 Adaptive Server 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]" [*ASA Database Administration Guide*, page 693].

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

About this chapter

This chapter describes the data types supported by Adaptive Server Anywhere.

Contents

Character data types

Function

For storing strings of letters, numbers and symbols.

Description

Adaptive Server Anywhere treats CHAR, VARCHAR, and LONG VARCHAR columns all as the same type. Values up to 254 characters are stored as short strings, with a preceding length byte. Any values that are longer than 255 bytes are considered long strings. Characters after the 255th byte are stored separately from the row containing the long string value.

There are several functions (see “SQL Functions” on page 91) that will ignore the part of any string past the 255th character. They are SOUNDEX, SIMILAR, and all of the date functions. Also, any arithmetic involving the conversion of a long string to a number will work on only the first 255 characters. It would be extremely unusual to run in to one of these limitations.

All other functions and all other operators work with the full length of long strings.

Character sets and code pages

Character data is placed in the database using the exact binary representation that is passed from the application. This usually means that character data is stored in the database with the binary representation of the current **code page**. The code page is the character set representation used by IBM-compatible personal computers. You can find documentation about code pages in the documentation for your operating system.

Most code pages are the same for the first 128 characters. If you use special characters from the top half of the code page (accented international language characters), you must be careful with your databases. In particular, if you copy the database to a machine that uses a different code page, those special characters will be retrieved from the database using the original code page representation. With the new code page, they will appear on the screen to be the wrong characters.

This problem also appears if you have two clients using the same multi-user server, but run with different code pages. Data inserted or updated by one client may appear incorrect to the other.

This problem also shows up if a database is used across platforms. PowerBuilder and many other Windows applications insert data into the database in the standard ANSI character set. If non-Windows applications attempt to use this data, they will not properly display or update the extended characters.

This problem is quite complex. If any of your applications use the extended characters in the upper half of the code page, make sure that all clients and all machines using the database use the same or a compatible code page.

Notes

Data type lengths of less than one are not allowed.

Compatibility

- The CHARACTER (n) alternative for CHAR is not supported in Adaptive Server Enterprise.
- Adaptive Server Anywhere does not support the NCHAR and NVARCHAR data types provided by Adaptive Server Enterprise.

CHAR data type [Character]

Function

Character data of maximum length *max-length* bytes.

Syntax

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

Usage

The default value of *max-length* is 1.

For strings up to 254 bytes in length, the storage requirement is the number of bytes in the string plus one additional byte. For longer strings, there is more overhead.

Strings of multi-byte characters can be held as the CHAR data type, but *max-length* is in bytes, not characters.

Parameters

max-length The maximum length in bytes of the string. The maximum size allowed is 32767.

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** Compatible with Adaptive Server Enterprise. In Adaptive Server Enterprise, the storage requirements for CHAR data types is always *max-length*. The maximum *max-length* for Adaptive Server Enterprise is 255.
- **Other database systems** In many other database-management systems, unlike Adaptive Server Anywhere, CHAR data types result in blank padding to the full length of the string. This means that they require *max-length* bytes of storage, regardless of the length of the actual string.

See also

- "CHARACTER VARYING (VARCHAR) data type [Character]" on page 55
- "LONG VARCHAR data type [Character]" on page 56

CHARACTER VARYING (VARCHAR) data type [Character]

Function

Same as CHAR.

Syntax

{ **VARCHAR** | **CHARACTER VARYING** } [**(** *max-length* **)**]

Usage

The default value of *max-length* is 1.

For strings up to 254 bytes in length, the storage requirements are the number of bytes in the string plus one additional byte. For longer strings, there is more overhead.

Strings of multi-byte characters can be held as the CHAR data type, but it is important to note that *max-length* is in bytes, not characters.

Parameters

max-length The maximum length of the string, in bytes. The maximum size allowed is 32767.

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** Compatible with Adaptive Server Enterprise. The maximum *max-length* for Adaptive Server Enterprise is 255.

See also

- "CHAR data type [Character]" on page 55
- "LONG VARCHAR data type [Character]" on page 56

LONG VARCHAR data type [Character]

Function

Arbitrary length character data.

Syntax

LONG VARCHAR

Usage

Arbitrary length strings. The maximum size is limited by the maximum size of the database file (currently 2 GB).

In addition to the length of the string itself, there is some additional overhead for storage.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- "CHAR data type [Character]" on page 55
- "CHARACTER VARYING (VARCHAR) data type [Character]" on page 55

TEXT data type [Character]

Function

This is a domain. It is implemented as a LONG VARCHAR allowing NULL.

Syntax

TEXT

Usage

Arbitrary length strings. The usage is as for LONG VARCHAR.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “LONG VARCHAR data type [Character]” on page 56

UNIQUEIDENTIFIERSTR data type [Character]

Function

This is a domain. It is implemented as a CHAR(36).

Syntax

UNIQUEIDENTIFIERSTR

Usage

This data type is used for remote data access, when mapping SQL Server uniqueidentifier columns.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “Data type conversions: Microsoft SQL Server” [*ASA SQL User’s Guide,* page 647]
- “STRTOUUID function [String]” on page 230

XML data type [Character]

Function

A domain, implemented as LONG VARCHAR.

Syntax

XML

Usage

XML is provided for storing XML documents in an Adaptive Server Anywhere database. 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” [*ASA SQL User’s Guide,* page 515].

Standards and compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- “LONG VARCHAR data type [Character]” on page 56

Numeric data types

Function

For storing numerical data.

Notes

- 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.*

- Before release 5.5, hexadecimal constants longer than four bytes were treated as string constants, and others were treated as integers. The new default behavior is to treat them as binary type constants. To use the historical behavior, set the TSQL_HEX_CONSTANTS database option to OFF.

- 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.

- You should avoid default precision and scale settings for NUMERIC and DECIMAL data types, because these are different between Adaptive Server Anywhere and Adaptive Server Enterprise. In Adaptive Server 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 Adaptive Server Enterprise, REAL is used for p less than or equal to 15, and DOUBLE for *p* greater than 15. For Adaptive Server 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]" [*ASA Database Administration Guide*, page 682] and "SCALE option [database]" [*ASA Database Administration Guide*, page 690].

BIGINT data type [Numeric]

Function

Integer requiring 8 bytes of storage.

Syntax

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

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- "INT or INTEGER data type [Numeric]" on page 61
- "TINYINT data type [Numeric]" on page 63
- "SMALLINT data type [Numeric]" on page 63

DECIMAL data type [Numeric]

Function

A decimal number with *precision* total digits and with *scale* of the digits after the decimal point.

Syntax

{ **DECIMAL** | **DEC** } [**(** *precision* [, *scale*] **)**]

Usage

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.

Parameters

precision An integer expression that specifies the number of digits in the expression. The default setting is 30.

scale An integer expression that specifies the number of digits after the decimal point. The default setting is 6.

The defaults can be changed by setting database options. For information, see "PRECISION option [database]" [*ASA Database Administration Guide*, page 682] and "SCALE option [database]" [*ASA Database Administration Guide*, page 690].

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "FLOAT data type [Numeric]" on page 60
- "REAL data type [Numeric]" on page 63
- "DOUBLE data type [Numeric]" on page 60

DOUBLE data type [Numeric]

Function

A double-precision floating-point number.

Syntax

DOUBLE [**PRECISION**]

Usage

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.

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "FLOAT data type [Numeric]" on page 60
- "REAL data type [Numeric]" on page 63
- "DECIMAL data type [Numeric]" on page 59

FLOAT data type [Numeric]

Function

A floating point number, which may be single or double precision.

Syntax

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

Usage

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 FLOATs require 4 bytes of storage, and double precision FLOATs 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.

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

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** You can tune the behavior of the FLOAT data type for compatibility with Adaptive Server Enterprise, using the "FLOAT_AS_DOUBLE option [compatibility]" [*ASA Database Administration Guide*, page 655].

See also

- "DECIMAL data type [Numeric]" on page 59
- "REAL data type [Numeric]" on page 63
- "DOUBLE data type [Numeric]" on page 60

INT or INTEGER data type [Numeric]

Function

Integer requiring 4 bytes of storage.

Syntax

[**UNSIGNED**] { **INT** | **INTEGER** }

Usage

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.

Standards and compatibility

- **SQL/92** Compatible with SQL/92. The UNSIGNED keyword is a vendor extension.
- **Sybase** The signed data type is compatible with Adaptive Server Enterprise. Adaptive Server Enterprise does not support the UNSIGNED data type.

See also

- "BIGINT data type [Numeric]" on page 58
- "TINYINT data type [Numeric]" on page 63
- "SMALLINT data type [Numeric]" on page 63

NUMERIC data type [Numeric]

Function

Same as DECIMAL.

Syntax

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

Usage

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.

Parameters

precision An integer expression that specifies the number of digits in the expression. The default value is 30.

scale An integer expression that specifies the number of digits after the decimal point. The default value is 6.

The defaults can be changed by setting database options. For information, see "PRECISION option [database]" [*ASA Database Administration Guide*, page 682] and "SCALE option [database]" [*ASA Database Administration Guide*, page 690].

Standards and compatibility

- **SQL/92** Compatible with SQL/92, if the SCALE option is set to zero.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "FLOAT data type [Numeric]" on page 60
- "REAL data type [Numeric]" on page 63
- "DOUBLE data type [Numeric]" on page 60

REAL data type [Numeric]

Function	A single-precision floating-point number stored in 4 bytes.
Syntax	**REAL**

Usage

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

Standards and compatibility

- **SQL/92** Compatible with SQL/92.
- **Sybase** Compatible with Adaptive Server Enterprise.

SMALLINT data type [Numeric]

Function	Integer requiring 2 bytes of storage.
Syntax	[**UNSIGNED**] **SMALLINT**

Usage

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.

Standards and compatibility

- **SQL/92** Compatible with SQL/92. The UNSIGNED keyword is a vendor extension.
- **Sybase** The signed data type is compatible with Adaptive Server Enterprise. Adaptive Server Enterprise does not support the UNSIGNED data type.

See also

- “INT or INTEGER data type [Numeric]” on page 61
- “TINYINT data type [Numeric]” on page 63
- “BIGINT data type [Numeric]” on page 58

TINYINT data type [Numeric]

Function	Unsigned integer requiring 1 byte of storage.

Syntax

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

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “BIGINT data type [Numeric]” on page 58
- “TINYINT data type [Numeric]” on page 63
- “SMALLINT data type [Numeric]” on page 63

Money data types

Function

For storing monetary data.

MONEY data type [Money]

Function

This data type is convenient for storing monetary data, and provides compatibility with the Adaptive Server Enterprise MONEY data type.

Syntax

MONEY

Usage

The MONEY data type is implemented as a domain, as NUMERIC(19,4), allowing NULL.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Monetary data types in Adaptive Server Anywhere are implemented as domains, and are primarily intended for compatibility with Adaptive Server Enterprise.

See also

- "SMALLMONEY data type [Money]" on page 65

SMALLMONEY data type [Money]

Function

This data type is convenient for storing monetary data that is not too large, and provides compatibility with the Adaptive Server Enterprise SMALLMONEY data type.

Syntax

SMALLMONEY

Usage

The SMALLMONEY data type is implemented in Adaptive Server Anywhere as a domain, as NUMERIC(10,4), allowing NULL.

The range for SMALLMONEY values is -214 748.3648 to 214 748.3647.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Monetary data types in Adaptive Server Anywhere are implemented as domains, and are primarily intended for compatibility with Adaptive Server Enterprise.

See also

- "MONEY data type [Money]" on page 65

BIT data type

Function	For storing Boolean values (0 or 1).
Syntax	**BIT**
Usage	By default, columns of BIT data type do not allow NULL. This behavior is different from other data types. You can explicitly allow NULL if desired. Allowed values are 0 and 1.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Date and time data types

Function For storing dates and times.

Sending dates and times to the database

Dates 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 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

Transact-SQL compatibility of string-to-date/time conversions

There are some differences in behavior between Adaptive Server Anywhere and Adaptive Server Enterprise, when 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, Adaptive Server Enterprise uses a default date of January 1, 1900, but Adaptive Server Anywhere uses the current date.

If the fraction portion of a time is less than 3 digits Adaptive Server Enterprise interprets the value differently depending on whether it was preceded by a period or a colon. If preceded by a colon, the value means thousandths of a second. If preceded by a period, one digit means tenths, two digits mean hundredths, and three digits mean thousandths. Adaptive Server Anywhere interprets the value the same way, regardless of the separator.

Examples

Adaptive Server Enterprise converts the values below as shown. The second line in each pair differs in the use of a colon rather than a period.

```
12:34:56.7 to 12:34:56.700
12:34:56:7 to 12:34:56.007
12.34.56.78 to 12:34:56.780
12.34.56:78 to 12:34:56.078
12:34:56.789 to 12:34:56.789
12:34:56:789 to 12:34:56.789
```

Adaptive Server Anywhere converts the milliseconds value in the manner that Adaptive Server Enterprise does for values preceded by a period, in both cases:

```
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 613.

☞ 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.

Comparing dates and times in the database

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

If you set the TRUNCATE_DATE_VALUES option to OFF, then the DATE data type also contains 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 wish 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.sss* 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. Adaptive Server 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 613). 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 option NEAREST_CENTURY 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

- We recommend that you always specify the year using the four-digit format.

  For more information about Y2K compliance issues, see "Year 2000 compliance" on page 85.

- 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 [Date and Time]

Function

A calendar date, such as a year, month and day.

Syntax

DATE

Usage

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

For historical reasons, a DATE column can also contain an hour and minute if the TRUNCATE_DATE_VALUES option is set to OFF. 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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **Sybase** Supported by Adaptive Server Enterprise 12.5.1 and later.

See also

- ♦ "DATE_FORMAT option [compatibility]" [*ASA Database Administration Guide,* page 646]
- ♦ "DATE_ORDER option [compatibility]" [*ASA Database Administration Guide,* page 648]
- ♦ "DATETIME data type [Date and Time]" on page 72
- ♦ "SMALLDATETIME data type [Date and Time]" on page 72
- ♦ "TIMESTAMP data type [Date and Time]" on page 73
- ♦ "TRUNCATE_DATE_VALUES option [database] (deprecated)" [*ASA Database Administration Guide,* page 697]

DATETIME data type [Date and Time]

Function

A domain, implemented as TIMESTAMP.

Syntax

DATETIME

Usage

DATETIME is provided primarily for compatibility with Adaptive Server Enterprise.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise. For an exception, see "Transact-SQL compatibility of string-to-date/time conversions" on page 67.

See also

- "DATE data type [Date and Time]" on page 71
- "SMALLDATETIME data type [Date and Time]" on page 72
- "TIMESTAMP data type [Date and Time]" on page 73

SMALLDATETIME data type [Date and Time]

Function

A domain, implemented as TIMESTAMP.

Syntax

SMALLDATETIME

Usage

SMALLDATETIME is provided primarily for compatibility with Adaptive Server Enterprise.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise. For an exception, see "Transact-SQL compatibility of string-to-date/time conversions" on page 67.

See also

- "DATE data type [Date and Time]" on page 71
- "DATETIME data type [Date and Time]" on page 72
- "TIMESTAMP data type [Date and Time]" on page 73

TIME data type [Date and Time]

Function

The time of day, containing hour, minute, second and fraction of a second.

Syntax

TIME

Usage

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.)

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Supported by Adaptive Server Enterprise 12.5.1 and later.

See also

- "TIMESTAMP data type [Date and Time]" on page 73

TIMESTAMP data type [Date and Time]

Function

The point in time, containing year, month, day, hour, minute, second and fraction of a second.

Syntax

TIMESTAMP

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- "TIME data type [Date and Time]" on page 72

Binary data types

Function	For storing binary data, including images and other information that is not interpreted by the database.

BINARY data type [Binary]

Function — Binary data of a specified maximum length (in bytes).

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

Usage — The default *max-length* is 1.

The maximum size allowed is 32767. The BINARY data type is identical to the CHAR data type except when used in comparisons. BINARY values are compared exactly while CHAR values are compared using the collation sequence of the database.

Parameters — *max-length* An integer expression that specifies the maximum length of the expression.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Adaptive Server Enterprise supports *max-length* up to 255.

See also

- "LONG BINARY data type [BINARY]" on page 74
- "VARBINARY data type [BINARY]" on page 76

LONG BINARY data type [BINARY]

Function — Arbitrary length binary data.

Syntax — **LONG BINARY**

Usage — The maximum size is 2 GB.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "BINARY data type [Binary]" on page 74
- "VARBINARY data type [BINARY]" on page 76

IMAGE data type [BINARY]

Function — LONG BINARY data allowing NULL.

Syntax — **IMAGE**

Usage

IMAGE is implemented in Adaptive Server Anywhere as a domain, as LONG BINARY allowing NULL. It is provided primarily for compatibility with Adaptive Server Enterprise.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

UNIQUEIDENTIFIER data type [Binary]

Function

Storage of UUID (also known as GUID) values.

Syntax

UNIQUEIDENTIFIER

Usage

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).

Adaptive Server 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.
- **Backwards compatibility** In databases created before version 9.0.2, UNIQUEIDENTIFIER was implemented as a user-defined data type using BINARY(16). The STRTOUUID and UUIDTOSTR functions were needed to convert the values between BINARY(16) and string representations. The STRTOUUID and UUIDTOSTR functions are not needed in databases created with version 9.0.2 or later, which support native UNIQUEIDENTIFIER data types.

To confirm that your database supports the native UNIQUEIDENTIFER data type, check that the UniqueIdentifier property in the Extended Information tab of the Sybase Central Database Properties sheet is **ON**, or that the following query returns **ON**:

```
select db_property('UniqueIdentifier')
```

See also

- "The NEWID default" [*ASA SQL User's Guide*, page 86]
- "NEWID function [Miscellaneous]" on page 185
- "UUIDTOSTR function [String]" on page 239
- "STRTOUUID function [String]" on page 230

VARBINARY data type [BINARY]

Function

Identical to BINARY.

Syntax

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

Usage

Variable length binary strings. The default value for *max-length* is 1.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "BINARY data type [Binary]" on page 74
- "LONG BINARY data type [BINARY]" on page 74

Domains

Function

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 Adaptive Server 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 full description of the syntax, see "CREATE DOMAIN statement" on page 347.

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 because CREATE DOMAIN is the syntax used in the draft SQL/3 standard.

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 the DBA, 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 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 Adaptive Server 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 Adaptive Server Anywhere, you can use the **sp_addtype** system procedure to add a domain, or you can use the CREATE DOMAIN statement. In Adaptive Server Enterprise, you must use **sp_addtype**.

Data type conversions

Type conversions can happen automatically, or they can be explicitly requested using the CAST or CONVERT function.

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.

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

The automatic data type conversion fails here. You can use the CAST or CONVERT function to force type conversions. For information about the CAST and CONVERT functions, see "Data type conversion functions" on page 93.

The following functions can also be used to force type conversions (see "SQL Functions" on page 91).

- **DATE(value)** Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.
- **STRING(value)** Equivalent to CAST(value AS LONG VARCHAR).

  ☞ For more information about the STRING function, see "STRING function [String]" on page 230.
- **VALUE+0.0** Equivalent to CAST(value AS DECIMAL).

☞ For more information about the CAST function, see "CAST function [Data type conversion]" on page 112.

Conversion when using comparison operators

When a comparison (such as =) is performed between arguments with different data types, one or both arguments must be converted so that the comparison is done using one data type. Sometimes it is preferable for you to explicitly convert one of the arguments using CAST or CONVERT.

Adaptive Server Anywhere uses the following rules to perform a comparison (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 one argument is TINYINT and the other is INTEGER, convert both to INTEGER and compare.

3. If one argument is TINYINT and the other is SMALLINT, convert both to SMALLINT and compare.

4. If one argument is UNSIGNED SMALLINT and the other is INTEGER, convert both to INTEGER and compare.

5. 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
   - DATE ➤ TIMESTAMP

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

6. 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.

7. If the data type of either argument is UNIQUEIDENTIFIER, convert both to UNIQUEIDENTIFIER and compare.

8. If one argument has CHARACTER data type and the other has BINARY data type, convert both to BINARY and compare.

9. If one argument has NUMERIC data type and the other has FLOAT, convert both to DOUBLE and compare.

10. Convert to NUMERIC and compare.

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

Notes

- You can override these 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.

  ☞ For more information, see "CAST function [Data type conversion]" on page 112.

- Rules 1, 6, and 10 may lead to conversions that fail.

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.lang.Long	INTEGER

Java type	SQL type
void	this[1]

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

[1]The method returns the object itself

Year 2000 compliance

The problem of handling dates, in particular year values in and beyond the year 2000, was a significant issue for the computer industry.

This section examines the year 2000 compliance of Adaptive Server Anywhere. It illustrates how date values are handled internally by Adaptive Server Anywhere, and how Adaptive Server Anywhere handles ambiguous date information, such as the conversion of a two digit year string value.

Users of Sybase Adaptive Server Anywhere and its predecessors can be assured that dates are handled and stored internally in a manner not adversely effected by the transition from the 20th century to the 21st century.

Consider the following measurements of Adaptive Server Anywhere year 2000 compliance:

- Adaptive Server 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 Adaptive Server Anywhere internal storage of dates explicitly includes the century portion of a year value.
- The operation of Adaptive Server Anywhere is unaffected by any return value, including the current date.
- Date values can always be output in full century format.

Many of the date–related topics summarized in this section are explained in greater detail in other parts of the documentation.

How dates are stored

Dates containing year values are used internally and stored in Adaptive Server 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 Adaptive Server Anywhere date and time data types see "Date and time data types" on page 67.

Sending and retrieving date values

Date values are stored within Adaptive Server Anywhere as either a DATE or TIMESTAMP data type, but they are passed to and retrieved from Adaptive Server Anywhere using one of the following methods:

- ♦ As a string, using any Adaptive Server Anywhere programming interface.
- ♦ As a TIMESTAMP structure, using ODBC.
- ♦ As a SQLDATETIME structure, using Embedded SQL.

A string containing a date value is considered unambiguous and is automatically converted to a DATE or TIMESTAMP data type without potential for misinterpretation if it is passed using the following format: *yyyy-mm-dd* (the "-" dash separator is one of several characters that are permitted).

For more information

Date formats other than *yyyy-mm-dd* can be used by setting the DATE_FORMAT database option. For more information, see "DATE_FORMAT option [compatibility]" [*ASA Database Administration Guide*, page 646].

For more information on unambiguous date formats, see "Using unambiguous dates and times" on page 69.

For more information on the ODBC TIMESTAMP structure, see the

Microsoft Open Database Connectivity SDK, or "Sending dates and times to the database" on page 67.

Used in the development of C programs, an embedded SQL SQLDATETIME structure's year value is a 16-bit signed integer.

For more information on the SQLDATETIME data type, see "Embedded SQL data types" [*ASA Programming Guide*, page 149].

Leap years

The year 2000 is a leap year, with an additional day in the month of February. Adaptive Server 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.

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

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

Adaptive Server 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 Adaptive Server 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.

Ambiguous string to date conversions

Adaptive Server 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 Adaptive Server 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. Adaptive Server 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 Adaptive Server 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;
```

Date to string conversions

Adaptive Server Anywhere provides several functions for converting Adaptive Server 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;
```

Adaptive Server 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.

CHAPTER 3

SQL Functions

About this chapter

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

NULL parameters
Unless otherwise stated, any function that receives NULL as a parameter returns NULL.

The chapter includes a grouping of functions by type, followed by an alphabetical list of functions.

Contents

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 109
- "COS function [Numeric]" on page 124
- "COVAR_POP function [Aggregate]" on page 126
- "COVAR_SAMP function [Aggregate]" on page 126
- "COUNT function [Aggregate]" on page 125
- "GROUPING function [Aggregate]" on page 160
- "LIST function [Aggregate]" on page 174
- "MAX function [Aggregate]" on page 180
- "MIN function [Aggregate]" on page 180
- "REGR_AVGX [Aggregate]" on page 204
- "REGR_AVGY [Aggregate]" on page 205
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_R2 [Aggregate]" on page 207
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "STDDEV function [Aggregate]" on page 227

- “STDDEV_POP function [Aggregate]” on page 227
- “STDDEV_SAMP function [Aggregate]” on page 228
- “SUM function [Aggregate]” on page 233
- “VAR_POP function [Aggregate]” on page 240
- “VAR_SAMP function [Aggregate]” on page 241
- “VARIANCE function [Aggregate]” on page 242

Rank functions

Rank 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 [Aggregate]” on page 129
- “DENSE_RANK function [Aggregate]” on page 142
- “PERCENT_RANK function [Aggregate]” on page 196
- “RANK function [Aggregate]” on page 203
- “ROW_NUMBER function [Aggregate]” on page 217

Data type conversion functions

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

Compatibility

- The Adaptive Server Anywhere **cast** function is not currently supported by Adaptive Server Enterprise.

List of functions

The following data type conversion functions are available:

- “CAST function [Data type conversion]” on page 112
- “CONVERT function [Data type conversion]” on page 121
- “HEXTOINT function [Data type conversion]” on page 161
- “INTTOHEX function [Data type conversion]” on page 169
- “ISDATE function [Data type conversion]” on page 170
- “ISNUMERIC function [Miscellaneous]” on page 171

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 130
- "DATEADD function [Date and time]" on page 131
- "DATEDIFF function [Date and time]" on page 131
- "DATEFORMAT function [Date and time]" on page 133
- "DATENAME function [Date and time]" on page 133
- "DATEPART function [Date and time]" on page 134
- "DATETIME function [Date and time]" on page 134
- "DAY function [Date and time]" on page 135
- "DAYNAME function [Date and time]" on page 135
- "DAYS function [Date and time]" on page 136
- "DOW function [Date and time]" on page 144
- "GETDATE function [Date and time]" on page 157
- "HOUR function [Date and time]" on page 162
- "HOURS function [Date and time]" on page 162
- "MINUTE function [Date and time]" on page 181
- "MINUTES function [Date and time]" on page 181
- "MONTH function [Date and time]" on page 183
- "MONTHNAME function [Date and time]" on page 183
- "MONTHS function [Date and time]" on page 184
- "NOW function [Date and time]" on page 189

- “QUARTER function [Date and time]” on page 201
- “SECOND function [Date and time]” on page 218
- “SECONDS function [Date and time]” on page 219
- “TODAY function [Date and time]” on page 235
- “WEEKS function [Date and time]” on page 243
- “YEAR function [Date and time]” on page 249
- “YEARS function [Date and time]” on page 250
- “YMD function [Date and time]” on page 251

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
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

Date Part	Abbreviation	Values
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. (Sunday = 1, . . . , Saturday = 7)

Java and SQL user-defined functions

There are two mechanisms for creating user-defined functions in Adaptive Server 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 362. 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" [*ASA SQL User's Guide*, page 655].

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 "A Java seminar" [*ASA Programming Guide*, page 59].

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 107
- ♦ "COALESCE function [Miscellaneous]" on page 115
- ♦ "COMPRESS function [String]" on page 116
- ♦ "CONFLICT function [Miscellaneous]" on page 118
- ♦ "DECOMPRESS function [String]" on page 139
- ♦ "DECRYPT function [String]" on page 140
- ♦ "ENCRYPT function [String]" on page 144
- ♦ "ERRORMSG function [Miscellaneous]" on page 145
- ♦ "ESTIMATE function [Miscellaneous]" on page 146
- ♦ "ESTIMATE_SOURCE function [Miscellaneous]" on page 147
- ♦ "EXPERIENCE_ESTIMATE function [Miscellaneous]" on page 153
- ♦ "EXPLANATION function [Miscellaneous]" on page 153
- ♦ "EXPRTYPE function [Miscellaneous]" on page 154
- ♦ "GET_IDENTITY function [Miscellaneous]" on page 155
- ♦ "GRAPHICAL_PLAN function [Miscellaneous]" on page 157
- ♦ "GRAPHICAL_ULPLAN function [Miscellaneous]" on page 158
- ♦ "GREATER function [Miscellaneous]" on page 159
- ♦ "HASH function [String]" on page 160
- ♦ "IDENTITY function [Miscellaneous]" on page 167
- ♦ "IFNULL function [Miscellaneous]" on page 168
- ♦ "INDEX_ESTIMATE function [Miscellaneous]" on page 168
- ♦ "ISNULL function [Data type conversion]" on page 170
- ♦ "LESSER function [Miscellaneous]" on page 173

- “LONG_ULPLAN function [Miscellaneous]” on page 178
- “NEWID function [Miscellaneous]” on page 185
- “NULLIF function [Miscellaneous]” on page 189
- “NUMBER function [Miscellaneous]” on page 190
- “PLAN function [Miscellaneous]” on page 198
- “REWRITE function [Miscellaneous]” on page 214
- “SHORT_ULPLAN function [Miscellaneous]” on page 220
- “SQLDIALECT function [Miscellaneous]” on page 226
- “TRACEBACK function [Miscellaneous]” on page 235
- “TRANSACTSQL function [Miscellaneous]” on page 235
- “VAREXISTS function [Miscellaneous]” on page 242
- “WATCOMSQL function [Miscellaneous]” on page 242

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 106
- “ACOS function [Numeric]” on page 106
- “ASIN function [Numeric]” on page 108
- “ATAN function [Numeric]” on page 108
- “ATN2 function [Numeric]” on page 109
- “CEILING function [Numeric]” on page 113
- “COS function [Numeric]” on page 124
- “COT function [Numeric]” on page 124
- “DEGREES function [Numeric]” on page 141
- “EXP function [Numeric]” on page 152
- “FLOOR function [Numeric]” on page 155

- “LOG function [Numeric]” on page 177
- “LOG10 function [Numeric]” on page 178
- “MOD function [Numeric]” on page 182
- “PI function [Numeric]” on page 197
- “POWER function [Numeric]” on page 199
- “RADIANS function [Numeric]” on page 202
- “RAND function [Numeric]” on page 202
- “REMAINDER function [Numeric]” on page 212
- “ROUND function [Numeric]” on page 216
- “SIGN function [Numeric]” on page 220
- “SIN function [Numeric]” on page 221
- “SQRT function [Numeric]” on page 227
- “TAN function [Numeric]” on page 234
- “TRUNCATE function [Numeric]” on page 236
- “TRUNCNUM function [Numeric]” on page 237

HTTP functions

HTTP functions facilitate the handling of HTTP requests within web services.

List of functions

The following functions are available:

- “HTML_DECODE function” on page 163
- “HTML_ENCODE function” on page 164
- “HTTP_DECODE function [HTTP]” on page 165
- “HTTP_ENCODE function [HTTP]” on page 165
- “HTTP_VARIABLE function [HTTP]” on page 167
- “NEXT_HTTP_HEADER function [HTTP]” on page 188
- “NEXT_HTTP_VARIABLE function [HTTP]” on page 188

String functions

String functions perform conversion, extraction, or manipulation operations on strings, or return information about strings.

When working in a multi-byte 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 107
- ♦ “BASE64_DECODE function [String]” on page 110
- ♦ “BASE64_ENCODE function [String]” on page 110
- ♦ “BYTE_LENGTH function [String]” on page 111
- ♦ “BYTE_SUBSTR function [String]” on page 111
- ♦ “CHAR function [String]” on page 113
- ♦ “CHARINDEX function [String]” on page 113
- ♦ “CHAR_LENGTH function [String]” on page 114
- ♦ “COMPARE function [String]” on page 115
- ♦ “COMPRESS function [String]” on page 116
- ♦ “CSCONVERT function [String]” on page 127
- ♦ “DECOMPRESS function [String]” on page 139
- ♦ “DECRYPT function [String]” on page 140
- ♦ “DIFFERENCE function [String]” on page 143
- ♦ “ENCRYPT function [String]” on page 144
- ♦ “HASH function [String]” on page 160
- ♦ “INSERTSTR function [String]” on page 169
- ♦ “LCASE function [String]” on page 172
- ♦ “LEFT function [String]” on page 172
- ♦ “LENGTH function [String]” on page 173
- ♦ “LOCATE function [String]” on page 176

- “LOWER function [String]” on page 179
- “LTRIM function [String]” on page 179
- “OPENXML function [String]” on page 191
- “PATINDEX function [String]” on page 196
- “REPEAT function [String]” on page 212
- “REPLACE function [String]” on page 213
- “REPLICATE function [String]” on page 213
- “RIGHT function [String]” on page 216
- “RTRIM function [String]” on page 218
- “SIMILAR function [String]” on page 221
- “SORTKEY function [String]” on page 222
- “SOUNDEX function [String]” on page 225
- “SPACE function [String]” on page 226
- “STR function [String]” on page 229
- “STRING function [String]” on page 230
- “STRTOUUID function [String]” on page 230
- “STUFF function [String]” on page 231
- “SUBSTRING function [String]” on page 232
- “TRIM function [String]” on page 236
- “UCASE function [String]” on page 238
- “UPPER function [String]” on page 238
- “UUIDTOSTR function [String]” on page 239
- “XMLAGG function [String]” on page 244
- “XMLCONCAT function [String]” on page 245
- “XMLELEMENT function [String]” on page 246
- “XMLFOREST function [String]” on page 247
- “XMLGEN function [String]” on page 248

System functions

System functions return system information.

List of functions

The following system functions are available:

- “CONNECTION_PROPERTY function [System]” on page 117
- “DATALENGTH function [System]” on page 130
- “DB_ID function [System]” on page 137
- “DB_NAME function [System]” on page 137
- “DB_EXTENDED_PROPERTY function [System]” on page 138
- “DB_PROPERTY function [System]” on page 139
- “EVENT_CONDITION function [System]” on page 148
- “EVENT_CONDITION_NAME function [System]” on page 149
- “EVENT_PARAMETER function [System]” on page 150
- “NEXT_CONNECTION function [System]” on page 186
- “NEXT_DATABASE function [System]” on page 187
- “PROPERTY function [System]” on page 200
- “PROPERTY_DESCRIPTION function [System]” on page 199
- “PROPERTY_NAME function [System]” on page 200
- “PROPERTY_NUMBER function [System]” on page 201

Compatibility

The following table displays the Adaptive Server Enterprise system functions and their status in Adaptive Server Anywhere:

Function	Status
`Col_length`	Implemented
`Col_name`	Implemented
`Curunreservedpgs`	Not implemented

Function	Status
`Data_pgs`	Not implemented
`Datalength`	Implemented
`Db_id`	Implemented
`Db_name`	Implemented
`Host_id`	Not implemented
`Host_name`	Not implemented
`Index_col`	Implemented
`Lct_admin`	Not implemented
`Object_id`	Implemented
`Object_name`	Implemented
`Proc_role`	Always returns 0
`Reserved_pgs`	Not implemented
`Rowcnt`	Not implemented
`Show_role`	Always returns NULL
`Suser_id`	Implemented
`Suser_name`	Implemented
`Tsequal`	Implemented

Function	Status
Used_pgs	Not implemented
User_id	Implemented
User_name	Implemented
Valid_name	Not implemented
Valid_user	Not implemented

Notes

- Some of the system functions are implemented in Adaptive Server 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

System function	Description
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 (server user name in Adaptive Server Enterprise)
Tsequal (*timestamp*, *timestamp2* **)**	Compares timestamp values to prevent update on a row that has been modified since it was selected
User_id([*user-name*] **)**	Returns an integer user identification number. This does not return the Adaptive Server Anywhere user ID
User_name([*user-id*] **)**	Returns the user ID (user name in Adaptive Server Enterprise)

Text and image functions

Text and image functions operate on text and image data types. Adaptive Server Anywhere supports only the textptr text and image function.

Compatibility

- Adaptive Server Anywhere does not currently support the Adaptive Server Enterprise **textvalid** function.

List of functions

The following text and image function is available:

- "TEXTPTR function [Text and image]" on page 234

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 92.

ABS function [Numeric]

Function

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/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 66.

```
SELECT ABS( -66 )
```

ACOS function [Numeric]

Function

Returns the arc-cosine, in radians, of a numeric expression.

Syntax

ACOS (*numeric-expression* **)**

Parameters

numeric-expression The cosine of the angle.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "ASIN function [Numeric]" on page 108
- "ATAN function [Numeric]" on page 108
- "ATN2 function [Numeric]" on page 109
- "COS function [Numeric]" on page 124

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 )
```

Example

The following statement returns the value 1.023945.

```
SELECT ACOS( 0.52 )
```

ARGN function [Miscellaneous]

Function

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.

Usage

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/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 6.

```
SELECT ARGN( 6, 1,2,3,4,5,6 )
```

ASCII function [String]

Function

Returns the integer ASCII value of the first byte in a string-expression.

Syntax

ASCII (*string-expression* **)**

Parameters

string-expression The string.

Usage

If the string is empty, then ASCII returns zero. Literal strings must be enclosed in quotes. If the database character set is multi-byte and the first character of the parameter string consists of more than one byte, the result is NULL.

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 90.

```
SELECT ASCII( 'Z' )
```

ASIN function [Numeric]

Function

Returns the arc-sine, in radians, of a number.

Syntax

ASIN (*numeric-expression* **)**

Parameters

numeric-expression The sine of the angle.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "ACOS function [Numeric]" on page 106
- "ATAN function [Numeric]" on page 108
- "ATN2 function [Numeric]" on page 109
- "SIN function [Numeric]" on page 221

Example

The following statement returns the value 0.546850.

```
SELECT ASIN( 0.52 )
```

ATAN function [Numeric]

Function

Returns the arc-tangent, in radians, of a number.

Syntax

ATAN (*numeric-expression* **)**

Parameters

numeric-expression The tangent of the angle.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "ACOS function [Numeric]" on page 106
- "ASIN function [Numeric]" on page 108
- "ATN2 function [Numeric]" on page 109
- "TAN function [Numeric]" on page 234

Example
The following statement returns the value **0.479519**.

```
SELECT ATAN( 0.52 )
```

ATN2 function [Numeric]

Function
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.

Standards and compatibility
- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** ATN2 is compatible with Adaptive Server Enterprise. ATAN2 is not supported by Adaptive Server Enterprise.

See also
- "ACOS function [Numeric]" on page 106
- "ASIN function [Numeric]" on page 108
- "ATAN function [Numeric]" on page 108
- "TAN function [Numeric]" on page 234

Example
The following statement returns the value 0.008666.

```
SELECT ATAN2( 0.52, 060 )
```

AVG function [Aggregate]

Function
Computes the average, for a set of rows, of a numeric-expression or of a set unique values.

Syntax
AVG (*numeric-expression* | **DISTINCT** *column-name* **)**

Parameters
numeric-expression The expression whose average is calculated over a set of rows.

DISTINCT column-name Computes the average of the unique values in *column-name*. This is of limited usefulness, but is included for completeness.

Usage
This average does not include rows where the *numeric expression* is the NULL value. Returns the NULL value for a group containing no rows.

Standards and compatibility
- **SQL/92** SQL/92 compatible.

- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "SUM function [Aggregate]" on page 233
- "COUNT function [Aggregate]" on page 125

Example

The following statement returns the value 49988.6.

```
SELECT AVG( salary ) FROM employee
```

BASE64_DECODE function [String]

Function

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.

Standards and compatibility

SQL/92 Vendor extension.

SQL/99 Vendor extension.

Sybase Not supported by Adaptive Server Enterprise.

See also

- "BASE64_ENCODE function [String]" on page 110

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]

Function

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.

Standards and compatibility

SQL/92 Vendor extension.

SQL/99 Vendor extension.

Sybase Not supported by Adaptive Server Enterprise.

See also

- "BASE64_DECODE function [String]" on page 110

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
```

BYTE_LENGTH function [String]

Function

Returns the number of bytes in a string.

Syntax

BYTE_LENGTH (*string-expression* **)**

Parameters

string-expression The string whose length is to be calculated.

Usage

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 multi-byte character set, the BYTE_LENGTH value differs from the number of characters returned by CHAR_LENGTH.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “CHAR_LENGTH function [String]” on page 114
- “DATALENGTH function [System]” on page 130
- “LENGTH function [String]” on page 173

Example

The following statement returns the value 12.

```
SELECT BYTE_LENGTH( 'Test Message' )
```

BYTE_SUBSTR function [String]

Function

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.

Usage

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "SUBSTRING function [String]" on page 232

Example

The following statement returns the value **Test**.

```
SELECT BYTE_SUBSTR( 'Test Message',1,4 )
```

CAST function [Data type conversion]

Function

Returns the value of an expression converted to a supplied data type.

Syntax

CAST (*expression* **AS** *data type* **)**

Parameters

expression The expression to be converted.

data type The target data type.

Usage

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.

Standards and compatibility

- ♦ **SQL/92** This function is SQL/92 compatible.
- ♦ **SQL/99** Core feature.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "CONVERT function [Data type conversion]" on page 121

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 cast into a single-character string.

```
SELECT CAST( 1 + 2 AS CHAR )
```

You can use the **CAST** function to shorten strings:

```
SELECT CAST( 'Surname' AS CHAR(5) )
```

CEILING function [Numeric]

Function	Returns the ceiling (smallest integer not less than) of a number.
Syntax	**CEILING (** *numeric-expression* **)**
Parameters	**numeric-expression** The number whose ceiling is to be calculated.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Compatible with Adaptive Server Enterprise.
See also	♦ "FLOOR function [Numeric]" on page 155
Example	The following statement returns the value 60.

```
SELECT CEILING( 59.84567 )
```

CHAR function [String]

Function	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.
Usage	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.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Compatible with Adaptive Server Enterprise.
Example	The following statement returns the value Y.

```
SELECT CHAR( 89 )
```

CHARINDEX function [String]

Function	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.

Usage 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 CHARINDEX returns the position of the first instance.

If the string being searched does not contain the other string, then CHARINDEX returns 0.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "SUBSTRING function [String]" on page 232

Example The statement

```
SELECT emp_lname, emp_fname
FROM employee
WHERE CHARINDEX('K', emp_lname ) = 1
```

returns the following values:

emp_lname	emp_fname
Klobucher	James
Kuo	Felicia
Kelly	Moira

CHAR_LENGTH function [String]

Function Returns the number of characters in a string.

Syntax **CHAR_LENGTH (** *string-expression* **)**

Parameters **string-expression** The string whose length is to be calculated.

Usage 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 multi-byte character set, the CHAR_LENGTH value differs from the number of bytes returned by BYTE_LENGTH.

Standards and compatibility

- **SQL/92** This function is SQL/92 compatible.

- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "BYTE_LENGTH function [String]" on page 111

Example

The following statement returns the value **8**.

```
SELECT CHAR_LENGTH( 'Chemical' )
```

COALESCE function [Miscellaneous]

Function

Returns the first non-NULL expression from a list.

Syntax

COALESCE (*expression*, *expression* [, . . .] **)**

Parameters

expression Any expression.

Standards and compatibility

- **SQL/92** SQL/92.
- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 34.

```
SELECT COALESCE( NULL, 34, 13, 0 )
```

COMPARE function [String]

Function

Allows you to directly compare two character strings based on alternate collation rules.

Syntax

COMPARE (
string-expression-1,
string-expression-2
[, *collation-name* | , *collation-id*] **)**

Parameters

string-expression-1 The first string expression.

string-expression-2 The second string expression.

The string expression may only contain characters that are encoded in the database's character set.

collation-name A string or a character variable that specifies the name of the sort order to use. For a list of valid collation names, see "SORTKEY function [String]" on page 222.

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. For more information, see "SORTKEY function [String]" on page 222.

If you do not specify a collation name or id, the default is Default Unicode multilingual.

Usage

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. COMPARE 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 will not compare equally.

If either *string-expression-1* or *string-expression-2* is null, the result is null.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "SORTKEY function [String]" on page 222

COMPRESS function [String]

Function

Compresses the string and returns a LONG BINARY.

Syntax

COMPRESS(*string-expression* [, *algorithm*]

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.

algorithm Currently, the only supported algorithm is **zip**, and zip is used if the algorithm is not specified.

Usage

The COMPRESS function returns a LONG BINARY value that is 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "DECOMPRESS function [String]" on page 139

Example

The following example uses the COMPRESS function to compress data from the product_xml_info table shown below.

prod_id	prod_info
301	`<product_info> <item_name>Tee Shirt </item_name> <quantity_left>54 </quantity_left> <description>Medium Orange Tee Shirt </description> </product_info>`
302	`<product_info> <item_name>Tee Shirt </item_name> <quantity_left>75 </quantity_left> <description>One size fits all Black Tee Shirt </description> </product_info>`

Obtain the compressed value for the prod_info column for product 302:

```
SELECT COMPRESS('prod_info')
FROM product_xml_info
WHERE prod_id = '302'
```

CONNECTION_PROPERTY function [System]

Function

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" [*ASA Database Administration Guide,* page 713].

integer-expression-2 The connection ID of the current database connection. The current connection is used if this argument is omitted.

Usage

The current connection is used if the second argument is omitted.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "Connection-level properties" [*ASA Database Administration Guide,* page 713]
- "PROPERTY_NUMBER function [System]" on page 201

Example

The following statement returns the number of prepared statements being maintained.

```
SELECT connection_property( 'PrepStmt' )
```

CONFLICT function [Miscellaneous]

Function

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.

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “CREATE TRIGGER statement” on page 419
- “Managing conflicts” [*SQL Remote User’s Guide*, page 120]

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 cons.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, Adaptive Server 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 cons.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 cons.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]

Function

Returns an expression converted to a supplied data type.

Syntax

CONVERT (*data type*, *expression* [, *format-style*] **)**

Parameters

data type The data type to which the expression will be converted.

expression The expression to be converted.

format-style For converting strings to date or time data types and vice versa, the *format-style* is a style code number that describes the date format string to be used. The values of the *format-style* argument have the following meanings:

Without century (yy)	With century (yyyy)	Output
-	0 or 100	Mmm dd yyyy hh:nn:ss:sss AM (or PM)
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]

Without century (yy)	**With century (yyyy)**	**Output**
7	107	Mmm dd, yy[yy]
8	108	hh:nn:ss
-	9 or 109	Mmm dd yyyy hh:nn:ss:sssAM (or PM)
10	110	mm-dd-yy[yy]
11	111	[yy]yy/mm/dd
12	112	[yy]yymmdd
13	113	dd Mmm yyy hh:nn:ss:sss (24 hour clock, Europe default + milliseconds, 4-digit year)
14	114	hh:nn:ss:sss (24 hour clock)
20	120	yyyy-mm-dd hh:nn:ss:sss (24-hour clock, ODBC canonical, 4-digit year)
21	121	yyyy-mm-dd hh:nn:ss.sss (24 hour clock, ODBC canonical with milliseconds, 4-digit year)

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]" [*ASA Database Administration Guide,* page 646].

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "CAST function [Data type conversion]" on page 112

Example

The following statements illustrate the use of format styles:

```
SELECT CONVERT( CHAR( 20 ), order_date, 104 )
FROM sales_order
```

order_date
16.03.2000
20.03.2000
23.03.2000
25.03.2000
…

```
SELECT CONVERT( CHAR( 20 ), order_date, 7 )
FROM sales_order
```

order_date
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]

Function 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.

Usage The function returns a numeric value. 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*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Aggregate functions" on page 92
- "COVAR_POP function [Aggregate]" on page 126
- "STDDEV_POP function [Aggregate]" on page 227

Example

The following example performs a correlation to discover whether age is associated with income level. This function returns the value 0.440226756459957.

```
SELECT CORR (salary, (year(now()) - year(birth_date))) FROM
        employee
```

COS function [Numeric]

Function

Returns the cosine of a number.

Syntax

COS (*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "ACOS function [Numeric]" on page 106
- "COT function [Numeric]" on page 124
- "SIN function [Numeric]" on page 221
- "TAN function [Numeric]" on page 234

Example

The statement

```
SELECT COS( 0.52 )
```

returns the value 0.86781.

COT function [Numeric]

Function

Returns the cotangent of a number.

Syntax

COT (*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "COS function [Numeric]" on page 124
- "SIN function [Numeric]" on page 221
- "TAN function [Numeric]" on page 234

Example

The following statement returns the value 1.74653.

```
SELECT COT( 0.52 )
```

COUNT function [Aggregate]

Function

Counts the number of rows in a group depending on the specified parameters.

Syntax

COUNT (
*
| *expression*
| **DISTINCT** { *expression* | *column-name* } **)**

Parameters

***** Returns the number of rows in each group.

expression Returns the number of rows in each group where the *expression is not the null value.*

DISTINCT expression or column-name Returns the number of different values in the expression, or the column with name *column-name*. Rows where the value is the NULL value are not included in the count.

Standards and compatibility

- **SQL/92** SQL/92 compatible.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "AVG function [Aggregate]" on page 109
- "SUM function [Aggregate]" on page 233

Example

The following statement returns each unique city, and the number of rows with that city value:

```
SELECT city , Count(*)
FROM employee
GROUP BY city
```

COVAR_POP function [Aggregate]

Function

Returns the population covariance of a set of number pairs.

Syntax

COVAR_POP (*dependent-expression*, *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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** (x) * **SUM** (x) / n) / n

where x represents the *dependent-expression* and y represents the *independent-expression*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "COVAR_SAMP function [Aggregate]" on page 126
- "SUM function [Aggregate]" on page 233

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(birth_date)))
FROM employee
```

COVAR_SAMP function [Aggregate]

Function

Returns the sample covariance of a set of number pairs.

Syntax

COVAR_SAMP (*dependent-expression*, *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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** (x) * **SUM** (y) / n) / (n-1)

where x represents the *dependent-expression* and y represents the *independent-expression*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "COVAR_POP function [Aggregate]" on page 126
- "SUM function [Aggregate]" on page 233

Example

The following example returns the value 74782.94600540561.

```
SELECT COVAR_SAMP (salary, (year(now()) - year(birth_date)))
FROM employee
```

CSCONVERT function [String]

Function

Converts strings between character sets.

Syntax

CSCONVERT (
string-expression,
'target-charset'
[, *'source-charset'*] **)**

Parameters

string-expression The string.

target-charset The destination character set. *Target-charset* can be one of the following:

- **os_charset** The character set used by the operating system.
- **db_charset** The character set used by the database.
- **any other supported character set label** You can specify any of the Adaptive Server Anywhere supported character set labels. For more information, see "Character set labels" [*ASA Database Administration Guide*, page 369].

source-charset The character set used by the original string-expression. The default is *db_charset*. *Source-charset-name* can be one of the following:

- **os_charset** The character set used by the operating system.
- **db_charset** The character set used by the database.
- **any other supported character set label** You can specify any of the Adaptive Server Anywhere supported character set labels. For more information, see "Character set labels" [*ASA Database Administration Guide*, page 369].

Standards and compatibility

- **SQL/92** SQL/92 compatible.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "Starting a database server using character set translation" [*ASA Database Administration Guide*, page 356]

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 filename 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 filename must be explicitly converted to the operating system's character set before the file can be accessed. Filenames 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 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 filenames. An external stored procedure takes a filename 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 filename 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 [Aggregate]

Function

Computes the relative position of one value among a group of rows. It returns a decimal value between 0 and 1.

Syntax

CUME_DIST ( )

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/OLAP feature T611
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “DENSE_RANK function [Aggregate]” on page 142
- “PERCENT_RANK function [Aggregate]” on page 196
- “RANK function [Aggregate]” on page 203
- “ROW_NUMBER function [Aggregate]” on page 217

Example

The following example returns a result set that provides a cumulative distribution of the salaries of employees who live in California.

```
SELECT dept_id, emp_lname, salary,
CUME_DIST () OVER (PARTITION BY dept_id
ORDER BY salary DESC) "Rank"
FROM employee
WHERE state IN (’CA’)
```

Here is the result set:

dept_id	emp_lname	salary	Rank
200	Savarino	72300.000	0.333333333333333
200	Clark	45000.000	0.666666666666667
200	Overbey	39300.000	1

DATALENGTH function [System]

Function

Returns the length in bytes of the underlying storage for the result of an expression.

Syntax

DATALENGTH (*expression* **)**

Parameters

expression The expression is usually a column name. If the expression is a string constant, it must be enclosed in quotes.

Usage

The return values of DATALENGTH are as follows:

Data type	DATALENGTH
SMALLINT	2
INTEGER	4
DOUBLE	8
CHAR	Length of the data
BINARY	Length of the data

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value **27**, the longest string in the company_name column.

```
SELECT following MAX( DATALENGTH( company_name ) )
FROM customer
```

DATE function [Date and time]

Function

Converts the expression into a date, and removes any hours, minutes or seconds.

Syntax

DATE (*expression* **)**

Parameters

expression The value to be converted to date format. The expression is usually a string.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 1999-01-02 as a date.

```
SELECT DATE( '1999-01-02 21:20:53' )
```

DATEADD function [Date and time]

Function

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** | **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/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

- The following statement returns the value: 1995-11-02 00:00:00.0.

```
SELECT dateadd( month, 102, '1987/05/02' )
```

DATEDIFF function [Date and time]

Function

Returns the interval between two dates.

Syntax

DATEDIFF (*date-part*, *date-expression-1*, *date-expression-2* **)**

date-part :
year | **quarter** | **month** | **week** | **day** | **hour** | **minute** | **second** | **millisecond**

Parameters

date-part Specifies the date-part in which the interval is to be measured.

☞ For more information about 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.

Usage

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.

DATEDIFF 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, DATEDIFF 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, DATEDIFF 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, DATEDIFF 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/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

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]

Function

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 on date format descriptions, see "DATE_FORMAT option [compatibility]" [*ASA Database Administration Guide,* page 646].

Usage

Any allowable date format can be used for the string-expression.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Year 2000 compliance
It is possible to use the DATEFORMAT function to produce a string with the year value represented by only two digits. This can cause problems with year 2000 compliance even though no error has occurred.

For more information on year 2000 compliance, please see "Year 2000 compliance" on page 85.

See also

- "DATE_FORMAT option [compatibility]" [*ASA Database Administration Guide,* page 646]

Example

The following statement returns the value Jan 01, 1989.

```
SELECT DATEFORMAT( '1989-01-01', 'Mmm dd, yyyy' )
```

DATENAME function [Date and time]

Function

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*.

Usage

DATENAME returns a string, even if the result is numeric, such as 23 for the day.

Standards and compatibility

- ♦ **SQL/92** Transact-SQL extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value May.

```
SELECT datename( month , '1987/05/02' )
```

DATEPART function [Date and time]

Function

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. The date must contain the *date-part* field.

Standards and compatibility

- ♦ **SQL/92** Transact-SQL extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 5.

```
SELECT datepart( month , '1987/05/02' )
```

DATETIME function [Date and time]

Function

Converts an expression into a timestamp.

Syntax

DATETIME (*expression* **)**

Parameters

expression The *expression* to be converted. It is generally a string. Attempts to convert numerical values return an error.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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]

Function

Returns an integer from 1 to 31 corresponding to the day of the month of a date.

Syntax

DAY (*date-expression* **)**

Parameters

date-expression The date.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 12.

```
SELECT DAY( '2001-09-12' )
```

DAYNAME function [Date and time]

Function

Returns the name of the day of the week from the a date.

Syntax

DAYNAME(*date-expression* **)**

Parameters

date-expression The date.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value Saturday.

```
SELECT DAYNAME ( '1987/05/02' )
```

DAYS function [Date and time]

Function

Given a single date, this function returns the number of days since 0000-02-29.

Given two dates, this function returns the integer number of days between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 131 instead for this purpose.

Given one date and an integer, it adds the integer number of days to the specified date. It is recommended that you use the "DATEADD function [Date and time]" on page 131 instead for this purpose.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

DAYS ignores hours, minutes, and seconds.

Syntax 1

DAYS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

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 on casting data types, see "CAST function [Data type conversion]" on page 112.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the integer 729 889.

```
SELECT DAYS( '1998-07-13 06:07:12' )
```

The following statement returns the current Julian day.

```
SELECT DAYS( CURRENT DATE ) + 1721119
```

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 00:00:00.0. 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_ID function [System]

Function

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 0 if asademo is the only running database:

```
SELECT DB_ID( 'asademo' )
```

The following statement returns the value 0 if executed against the only running database.

```
SELECT DB_ID()
```

DB_NAME function [System]

Function

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.

Usage

If no database ID is supplied, the name of the current database is returned.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The statement returns the database name asademo, when executed against the sample database as the sole database on the server.

```
SELECT DB_NAME( 0 )
```

DB_EXTENDED_PROPERTY function [System]

Function

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.

property_name The database property name.

database_id The database ID number, as returned by DB_ID. Typically, the database name is used.

database_name The name of the database, as returned by DB_NAME.

property_specific_argument The optional property-specific string parameter associated with FileSize and FreePages properties. If you provide a property specific argument for a property that does not require one (for example, any property other than FileSize or FreePages), the item is ignored. For a list of properties, see "Database-level properties" [*ASA Database Administration Guide,* page 733].

Usage

Returns a string. The current database is used if the second argument is omitted.

Db_extended_property() is similar to db_property() 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. Calling **db_extended_property(x)** is equivalent to **calling db_property(x)**.

Standards and compatibility

- **SQL/92** Vendor extension.

  SQL/99 Vendor extension.

  Sybase Not supported by Adaptive Server Enterprise.

See also

- "DB_ID function [System]" on page 137

- ♦ "DB_NAME function [System]" on page 137
- ♦ "Database-level properties" [*ASA Database Administration Guide*, page 733]

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')
```

DB_PROPERTY function [System]

Function

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 DB_ID. Typically, the database name is used.

database_name The name of the database, as returned by DB_NAME.

Usage

Returns a string. The current database is used if the second argument is omitted.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "DB_ID function [System]" on page 137
- ♦ "DB_NAME function [System]" on page 137
- ♦ "Database-level properties" [*ASA Database Administration Guide*, page 733]

Example

The following statement returns the page size of the current database, in bytes.

```
SELECT DB_PROPERTY( 'PAGESIZE' )
```

DECOMPRESS function [String]

Function

Decompresses the string and returns a LONG BINARY value.

Syntax

DECOMPRESS(*string-expression* [, *algorithm*] **)**

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.

algorithm The default value is **zip**. Currently, this is the only available algorithm.

Usage

The DECOMPRESS function returns a LONG BINARY value. This function can be used to decompress a value that was compressed using the COMPRESS function.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "COMPRESS function [String]" on page 116

Example

The following example uses the DECOMPRESS function to decompress a value that was compressed by the COMPRESS function.

```
INSERT INTO product_xml_info
VALUES ('400', COMPRESS('<product_info>
 <item_name>Baseball Cap</item_name>
 <quantity_left>112</quantity_left>
 <description>One size fits all Black Baseball Cap</description>
</product_info>'))
```

Because the original value was of type XML, the CAST function is used to convert the LONG BINARY returned by the DECOMPRESS function to the XML data type.

```
  SELECT CAST (DECOMPRESS( prod_info ) AS XML)
  FROM product_xml_info
  WHERE prod_id = '400'
```

DECRYPT function [String]

Function

Decrypts the string using the supplied key and returns a LONG BINARY.

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 required to decrypt the *string-expression*. This must be the same encryption key that was used to encrypt the *string-expression* in order 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 it was encrypted with. Currently, AES is the only supported algorithm and it is used by default. AES is a block encryption algorithm chosen as the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

Usage

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.

In order 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "ENCRYPT function [String]" on page 144
- "Encrypting portions of a database" [*SQL Anywhere Studio Security Guide*, page 20]

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]

Function

Converts a number from radians to degrees.

Syntax

DEGREES (*numeric-expression* **)**

Parameters

numeric-expression An angle in radians.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 29.793805.

```
SELECT DEGREES( 0.52 )
```

DENSE_RANK function [Aggregate]

Function

Calculates the rank of a value in a partition. In the case of tied values, DENSE_RANK does not leave gaps in the ranking sequence.

Syntax

DENSE_RANK ( )

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/OLAP feature T611
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CUME_DIST function [Aggregate]" on page 129
- "PERCENT_RANK function [Aggregate]" on page 196
- "RANK function [Aggregate]" on page 203
- "ROW_NUMBER function [Aggregate]" on page 217

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 DENSE_RANK does not leave gaps in the ranks.

```
SELECT dept_id, emp_lname, salary, state,
DENSE_RANK () OVER (ORDER BY salary DESC) "Rank"
FROM employee
WHERE state IN ('NY','UT')
```

Here is the result set:

emp_lname	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

emp_lname	salary	state	Rank
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]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "SOUNDEX function [String]" on page 225

Example

The following statement returns the value 3.

```
SELECT DIFFERENCE( 'test', 'chest' )
```

DOW function [Date and time]

Function

Returns a number from 1 to 7 representing the day of the week of a date, with Sunday=1, Monday=2, and so on.

Syntax

DOW (*date-expression* **)**

Parameters

date-expression The date.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 5.

```
SELECT DOW( ’1998-07-09’ )
```

ENCRYPT function [String]

Function

Encrypts the specified values using the supplied encryption key and returns a LONG BINARY.

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 in order 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. We recommend that you choose a value for your key that is at least 16 characters long, contains a mix of upper and lower case, 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 The algorithm used to encrypt the data. This parameter is optional, but data must be encrypted and decrypted using the same algorithm. Currently, AES is the only supported algorithm and it is used by default. AES is a block encryption algorithm chosen as the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

Usage

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. In order 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "DECRYPT function [String]" on page 140
- "Encrypting portions of a database" [*SQL Anywhere Studio Security Guide*, page 20]

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]

Function

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).

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "Error messages indexed by SQLSTATE" [*ASA Error Messages*, page 36]
- ♦ "Error messages indexed by Adaptive Server Anywhere SQLCODE" [*ASA Error Messages*, page 2]

Example

The following statement returns the error message for SQLCODE -813.

```
select errormsg( -813 )
```

ESTIMATE function [Miscellaneous]

Function

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.

relation-string The comparison operator used for the comparison, enclosed in single quotes; the default is '='.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "INDEX_ESTIMATE function [Miscellaneous]" on page 168
- ♦ "ESTIMATE_SOURCE function [Miscellaneous]" on page 147

Example

The following statement returns the percentage of emp_id values estimated to be greater than 200. The precise value depends on the actions you have carried out on the database.

```
SELECT FIRST ESTIMATE( emp_id, 200, '>' )
   FROM employee
```

ESTIMATE_SOURCE function [Miscellaneous]

Function

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. This is optional.

relation-string The comparison operator used for the comparison, enclosed in single quotes. The default is equality (=).

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]" [*ASA Database Administration Guide*, page 700].

- **Computed** is returned when statistics are computed by the optimizer based on other information. For example, Adaptive Server 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 Adaptive Server 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "ESTIMATE function [Miscellaneous]" on page 146
- "INDEX_ESTIMATE function [Miscellaneous]" on page 168

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( emp_id, 200, '>' )
FROM employee
```

EVENT_CONDITION function [System]

Function

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	
RemainingValues	integer	GlobalAutoincrement	The number of remaining values
TempFreePercent	n/a	TempDiskSpace	
TempFreeSpace	Mb	TempDiskSpace	
TempSize	Mb	GrowTemp	

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CREATE EVENT statement" on page 351

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]

Function

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.

Usage

You can use EVENT_CONDITION_NAME to obtain a list of all EVENT_CONDITION arguments by looping over integers until the function returns NULL.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CREATE EVENT statement" on page 351

EVENT_PARAMETER function [System]

Function

Provides context information for event handlers.

Syntax

EVENT_PARAMETER (*context-name* **)**

context-name:
  'AppInfo'
| **'ConnectionID'**
| **DisconnectReason**
| **'EventName'**
| **'Executions'**
| **'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 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.
- ♦ **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

Context-name value	Valid system event types
Executions	all
NumActive	all
TableName	GlobalAutoincrement
User	BackupEnd, “Connect”, ConnectFailed, “Disconnect”, GlobalAutoincrement, “RAISERROR”, user events

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));
```

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “EVENT_CONDITION function [System]” on page 148
- “CREATE EVENT statement” on page 351
- “TRIGGER EVENT statement” on page 641

EXP function [Numeric]

Function

Returns the exponential function, e to the power of a number.

Syntax

EXP (*numeric-expression* **)**

Parameters

numeric-expression The exponent.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The statement returns the value 3269017.372.

```
SELECT EXP( 15 )
```

EXPERIENCE_ESTIMATE function [Miscellaneous]

Function

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, enclosed in single quotes; the default is ‘=‘.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “ESTIMATE function [Miscellaneous]” on page 146

Example

The following statement returns NULL.

```
SELECT DISTINCT EXPERIENCE_ESTIMATE( emp_id, 200, ’>’ )
FROM employee
```

EXPLANATION function [Miscellaneous]

Function

Returns the short plan optimization strategy of a SQL statement, as a string.

Syntax

EXPLANATION (
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.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "PLAN function [Miscellaneous]" on page 198
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 157
- "GRAPHICAL_ULPLAN function [Miscellaneous]" on page 158
- "LONG_ULPLAN function [Miscellaneous]" on page 178
- "SHORT_ULPLAN function [Miscellaneous]" on page 220

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT EXPLANATION(
   'SELECT * FROM department WHERE dept_id > 100' )
```

This information can help you decide which indexes to add or how to structure your database for better performance.

The following statement returns a string containing the short form of the textual plan for an INSENSITIVE cursor over the query 'select * from department where . . . .'.

```
SELECT EXPLANATION(
   'SELECT * FROM department WHERE dept_id > 100',
   'insensitive',
   'read-only' )
```

In Interactive SQL, you can view the plan for any SQL statement on the Plan tab in the Results pane.

EXPRTYPE function [Miscellaneous]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "SQL Data Types" on page 53

Example

The following statement returns `smallint` when executed against the sample database.

```
SELECT EXPRTYPE(
   'SELECT line_id FROM sales_order_items', 1 )
```

FLOOR function [Numeric]

Function

Returns the floor of (largest integer not greater than) a number.

Syntax

FLOOR (*numeric-expression* **)**

Parameters

numeric- expression The number, usually a float.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "CEILING function [Numeric]" on page 113

Example

Value	FLOOR (Value)
123	123
123.45	123
–123.45	–124

GET_IDENTITY function [Miscellaneous]

Function

Allocates values to an autoincrement column. This is an alternative to using autoincrement to generate numbers.

Syntax

GET_IDENTITY ([*owner*.] *table-name* [, *num_to_alloc*],. . . **)**

Parameters

num_to_allocate Default is 1.

Usage

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.

If *num_to_alloc* is greater than 1, the function also reserves the remaining values. The next allocation uses the current number plus the value of *num_to_alloc*. This allows the application to execute get_identity less frequently.

No COMMIT is required after executing get_identity, 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 get_identity, as in the example.

GET_IDENTITY is a non-deterministic function. Successive calls to GET_IDENTITY may return different values. The query optimizer does not cache the results of the GET_IDENTITY function.

☞ For more information about non-deterministic functions, see "Function caching" [*ASA SQL User's Guide*, page 449].

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CREATE TABLE statement" on page 407
- "ALTER TABLE statement" on page 294
- "NUMBER function [Miscellaneous]" on page 190

Example

The following statement makes three calls to the GET_IDENTITY function:

```
SELECT GET_IDENTITY('T1'),
   GET_IDENTITY('T2',10),
   GET_IDENTITY('T3',5)
```

GETDATE function [Date and time]

Function

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.

The information the GETDATE function returns is equivalent to the information returned by the NOW function and the CURRENT TIMESTAMP special value.

Syntax

GETDATE ()

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "NOW function [Date and time]" on page 189
- "CURRENT TIMESTAMP special value" on page 34

Example

The following statement returns the system date and time.

```
SELECT GETDATE( )
```

GRAPHICAL_PLAN function [Miscellaneous]

Function

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.

statistics-level An integer. *Statistics-level* can be one of the following values:

Value	Description
0	Include optimizer estimates only. (default)
1	Include actual summary statistics from execution.
2	Include detailed actual statistics.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "PLAN function [Miscellaneous]" on page 198
- "EXPLANATION function [Miscellaneous]" on page 153
- "GRAPHICAL_ULPLAN function [Miscellaneous]" on page 158
- "LONG_ULPLAN function [Miscellaneous]" on page 178
- "SHORT_ULPLAN function [Miscellaneous]" on page 220

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 department WHERE dept_id > 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 department WHERE ....'. 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 department WHERE dept_id > 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.

GRAPHICAL_ULPLAN function [Miscellaneous]

Function

Returns the UltraLite plan optimization strategy of a SQL statement in XML format, as a string. The UltraLite plan does not include statistics.

For some queries, the execution plan for UltraLite may differ from the plan selected for Adaptive Server Anywhere.

Syntax

GRAPHICAL_ULPLAN (*string-expression* **)**

Parameters

string-expression The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "PLAN function [Miscellaneous]" on page 198
- "EXPLANATION function [Miscellaneous]" on page 153
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 157
- "LONG_ULPLAN function [Miscellaneous]" on page 178
- "SHORT_ULPLAN function [Miscellaneous]" on page 220

Example

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_ULPLAN(
   'select * from department where dept_id > 100' );
OUTPUT TO ulplan.xml
FORMAT FIXED
```

To display the plan, open the *ulplan.xml* file in Interactive SQL.

As an alternative, you can view the plan for any SQL statement on the UltraLite Plan tab in Interactive SQL, choose File ➤ Save, and change the file type to xml. To change the type of plan that is displayed, choose Tools ➤ Options and open the Plan tab.

GREATER function [Miscellaneous]

Function

Returns the greater of two parameter values. If the parameters are equal, the first is returned.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "LESSER function [Miscellaneous]" on page 173

Example

The following statement returns the value 10.

```
SELECT GREATER( 10,5 ) FROM dummy
```

GROUPING function [Aggregate]

Function

Identifies whether a column in a ROLLUP 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 with the ROLLUP keyword. The function identifies subtotal rows added to the result set by a ROLLUP 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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** SQL/foundation feature outside of core SQL.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "Using ROLLUP" [*ASA SQL User's Guide*, page 335]
- ♦ "SELECT statement" on page 597

Example

See "Using ROLLUP" [*ASA SQL User's Guide*, page 335].

HASH function [String]

Function

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* can be one of **md5** or **sha1**. By default, the MD5 algorithm is used.

Usage

When the MD5 algorithm is specified, this function returns a 32 character VARCHAR string. When SHA-1 is used, the function returns a 40 character VARCHAR string.

Using a hash converts the value to a byte sequence that is unique to each value passed to the function.

> **Caution**
> *Both MD5 and SHA-1 are one-way hashes. It is not possible to re-create the original string from the hash.*

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following example creates a table called user_info that stores information about the users of an application, including their user ID and password.

```
CREATE TABLE user_info (
   emp_id    INTEGER NOT NULL PRIMARY KEY,
   user_name    CHAR(80),
   user_pwd    CHAR(80) );
INSERT INTO user_info VALUES ( '1', 's_phillips', '8uKj2s');
```

Generate the hashed form of the password using the SHA-1 algorithm:

```
SELECT HASH(user_pwd, 'sha1')
FROM user_info
WHERE emp_id = '1';
```

HEXTOINT function [Data type conversion]

Function

Returns the decimal integer equivalent of a hexadecimal string.

Syntax

HEXTOINT (*hexadecimal-string* **)**

Parameters

hexadecimal-string The string to be converted to an integer.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

See also

- ♦ "INTTOHEX function [Data type conversion]" on page 169

Example

The following statement returns the value 420.

```
SELECT HEXTOINT ( '1A4' )
```

HOUR function [Date and time]

Function

Returns a number from 0 to 23 corresponding to the hour component of a datetime.

Syntax

HOUR (*datetime-expression* **)**

Parameters

datetime-expression The datetime.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 21:

```
SELECT HOUR( '1998-07-09 21:12:13' )
```

HOURS function [Date and time]

Function

Given two timestamps, this function returns the integer number of hours between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 131 instead for this purpose.

Given a single date, this function returns the number of hours since 0000-02-29 00:00:00.

Given one date and an integer, it adds the integer number of hours to the specified timestamp. It is recommended that you use the "DATEADD function [Date and time]" on page 131 instead for this purpose.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

Syntax 1

HOURS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

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 on casting data types, see "CAST function [Data type conversion]" on page 112.

Standards and compatibility

- **SQL/92** Vendor extension.

- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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 17 517 342.

```
SELECT HOURS( ’1998-07-13 06:07:12’ )
```

The following statements return the datetime 1999-05-13 02:05:07.0. 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

Function

Decodes special character entities that appear in HTML literal strings.

Syntax

HTML_DECODE (*string* **)**

Parameters

string Arbitrary literal string used in an HTML document.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Usage

This function returns the string argument after making the following set of substitutions:

Characters	Substitution
&quot;	&quot;
&#39;	&#39;
&amp;	&
&lt;	<
&gt;	>
&#x*nn*;	character *nn*

See also

- "HTML_ENCODE function" on page 164
- "HTTP_DECODE function [HTTP]" on page 165

Example

HTML_ENCODE function

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Usage

This function returns the string argument after making the following set of substitutions:

Characters	Substitution
"	&quot;
'	&#39;
&	&amp;
<	&lt;
>	&gt;
codes *nn* less than 0x20	&#x*nn*;

See also

- "HTML_DECODE function" on page 163
- "HTTP_ENCODE function [HTTP]" on page 165

Example

HTTP_DECODE function [HTTP]

Function Decodes special characters within strings for use with HTTP.

Syntax **HTTP_DECODE (** *string* **)**

Parameters **string** Arbitrary string to be used in an HTTP request.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Usage 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 165
- "HTML_DECODE function" on page 163

Example

HTTP_ENCODE function [HTTP]

Function Encodes special characters in strings for use with HTTP.

Syntax **HTTP_ENCODE (** *string* **)**

Parameters **string** Arbitrary string to be used in an HTTP request.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Usage 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

Character	Substitution
&	%26
,	%2C
;	%3B
<	%3C
>	%3E
[	%5B
\	%5C
]	%5D
‘	%60
{	%7B
|	%7C
}	%7D

See also

- “HTTP_DECODE function [HTTP]” on page 165
- “HTML_ENCODE function” on page 164

Example

HTTP_HEADER function [HTTP]

Function Gets the value of an HTTP header.

Syntax **HTTP_HEADER (** *field-name* **)**

Parameters **field-name** The name of an HTTP header field.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Usage This function returns the value of the named HTTP header field. It is used when processing an HTTP request via a web service.

See also

- “HTTP_VARIABLE function [HTTP]” on page 167
- “NEXT_HTTP_HEADER function [HTTP]” on page 188
- “NEXT_HTTP_VARIABLE function [HTTP]” on page 188

HTTP_VARIABLE function [HTTP]

Function	Gets the value of an HTTP variable.
Syntax	**HTTP_VARIABLE (** *var-name* [[, *instance*] , *header-field*] **)**
Parameters	**var-name** The name of the 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. **header-field** In a multi-part request, a header field field name associated with the named field.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not supported by Adaptive Server Enterprise.
Usage	This function returns the value of the named HTTP variable. It is used when processing an HTTP request within a web service.
See also	♦ "HTTP_HEADER function [HTTP]" on page 166 ♦ "NEXT_HTTP_HEADER function [HTTP]" on page 188 ♦ "NEXT_HTTP_VARIABLE function [HTTP]" on page 188

IDENTITY function [Miscellaneous]

Function	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.
Usage	For a description of how to use the IDENTITY function, see the "NUMBER function [Miscellaneous]" on page 190.
Standards and compatibility	♦ **SQL/92** Transact-SQL extension. ♦ **SQL/99** Transact-SQL extension. ♦ **Sybase** Offers similar behavior to that of Adaptive Server Enterprise.
See also	♦ "NUMBER function [Miscellaneous]" on page 190

Example

The following statement returns a sequentially-numbered list of employees.

```
SELECT IDENTITY( 10 ), emp_lname FROM employee
```

IFNULL function [Miscellaneous]

Function

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.

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.

Standards and compatibility

- ♦ **SQL/92** Transact-SQL extension.
- ♦ **SQL/99** Transact-SQL extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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]

Function

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 *relation-string* must be a comparison operator enclosed in single quotes; the default is ‘=‘.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "ESTIMATE function [Miscellaneous]" on page 146
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 147

Example

The following statement returns the value 81.304607.

```
SELECT FIRST ESTIMATE( emp_id, 200, '>' )
FROM employee
```

INSERTSTR function [String]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "STUFF function [String]" on page 231

Example

The following statement returns the value backoffice.

```
SELECT INSERTSTR( 0, 'office ', 'back' )
```

INTTOHEX function [Data type conversion]

Function

Returns a string containing the hexadecimal equivalent of an integer.

Syntax

INTTOHEX (*integer-expression* **)**

Parameters

integer-expression The integer to be converted to hexadecimal.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.

- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "HEXTOINT function [Data type conversion]" on page 161

Example

The following statement returns the value 9c:

```
SELECT INTTOHEX( 156 )
```

ISDATE function [Data type conversion]

Function

Tests if a string argument can be converted to a date. If a conversion is possible, the function returns 1; otherwise, 0 is returned. If the argument is null, 0 is returned.

Syntax

ISDATE (*string* **)**

Parameters

string The string to be analyzed to determine if the string represents a valid date.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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,birthdate,height_in_cms
      from MyData
      where isdate(birth_date)=1
  and isnumeric(height_in_cms)=1;
   commit;
   drop table MyData;
```

ISNULL function [Data type conversion]

Function

Returns the first non-NULL expression in the parameter list.

Syntax

ISNULL (*expression*, *expression* [, . . .] **)**

Parameters

expression An expression to be tested against NULL.

At least two expressions must be passed into the function.

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise, except that Adaptive Server Enterprise allows only two expressions.

See also

- "COALESCE function [Miscellaneous]" on page 115

Example

The following statement returns the value –66.

```
SELECT ISNULL( NULL ,-66, 55, 45, NULL, 16 )
```

ISNUMERIC function [Miscellaneous]

Function

Tests if a string argument can be converted to a numeric. If a conversion is possible, the function returns 1; otherwise, 0 is returned. If the argument is null, 0 is returned.

Syntax

ISNUMERIC (*string* **)**

Parameters

string The string to be analyzed to determine if the string represents a valid numeric value.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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,birthdate,height_in_cms
      from MyData
      where isdate(birth_date)=1
  and isnumeric(height_in_cms)=1;
   commit;
   drop table MyData;
```

LCASE function [String]

Function

Converts all characters in a string to lower case.

Syntax

LCASE (*string-expression* **)**

Parameters

string-expression   The string to be converted to lower case.

Standards and compatibility

- **SQL/92**   Vendor extension.
- **SQL/99**   Vendor extension.
- **Sybase**   LCASE is not supported in Adaptive Server Enterprise; you can use LOWER to get the same functionality.

See also

- "LOWER function [String]" on page 179
- "UCASE function [String]" on page 238
- "UPPER function [String]" on page 238

Example

The following statement returns the value lower case.

```
SELECT LCASE( 'LOWER CasE' )
```

LEFT function [String]

Function

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.

Usage

If the string contains multi-byte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "RIGHT function [String]" on page 216
- "International Languages and Character Sets" [*ASA Database Administration Guide*, page 319]

Example

The following statement returns the value choco.

```
SELECT LEFT( 'chocolate', 5 )
```

LENGTH function [String]

Function

Returns the number of characters in the specified string.

Syntax

LENGTH (*string-expression* **)**

Parameters

string-expression The string.

Usage

If the string contains multi-byte characters, and the proper collation is being used, LENGTH returns the number of characters, not the number of bytes. If string is of BINARY data type, the LENGTH function behaves as BYTE_LENGTH.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "BYTE_LENGTH function [String]" on page 111
- "International Languages and Character Sets" [*ASA Database Administration Guide*, page 319]

Example

The following statement returns the value 9.

```
SELECT LENGTH( 'chocolate' )
```

LESSER function [Miscellaneous]

Function

Returns the lesser of two parameter values. If the parameters are equal, the first is returned.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "GREATER function [Miscellaneous]" on page 159

Example

The following statement returns the value 5.

```
SELECT LESSER( 10,5 ) FROM dummy
```

LIST function [Aggregate]

Function

Returns a comma-separated list of values.

Syntax

LIST (
{ *string-expression* | **DISTINCT** *column-name* }
[, *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 column-name 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.

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Examples

The following statement returns the value 48 Kennedy Court, 54 School Street.

```
SELECT LIST( street ) FROM employee
WHERE emp_fname = 'Thomas'
```

The following statement lists employee IDs. Each row in the result set contains a comma-separated list of employee IDs for a single department.

```
SELECT LIST( emp_id )
FROM employee
GROUP BY dept_id
```

LIST(emp_id)
102,105,160,243,247,249,266,27,. . .
129,195,299,467,641,667,690,85,. . .
148,390,586,757,879,1293,1336,. . .
184,207,318,409,591,888,992,10,. . .
191,703,750,868,921,1013,1570,. . .

The following statement sorts the employee IDs by the last name of the employee:

```
SELECT LIST( emp_id ORDER BY emp_lname ) AS "Sorted IDs"
FROM EMPLOYEE
GROUP BY dept_id
```

Sorted IDs
160,105,1250,247,266,249,445,. . .
1039,129,1142,195,667,1162,902,. . .
1336,879,586,390,757,148,1483,. . .
1751,591,1062,1191,992,888,318,. . .
1013,191,750,921,868,1658,703,. . .

The following statement returns semicolon-separated lists. Note the position of the ORDER BY clause and the list separator:

```
SELECT LIST( emp_id, ';' ORDER BY emp_lname ) AS "Sorted IDs"
FROM EMPLOYEE
GROUP BY dept_id
```

Sorted IDs
160;105;1250;247;266;249;445;...
1039;129;1142;195;667;1162;902;...
1336;879;586;390;757;148;1483;...
1751;591;1062;1191;992;888;318;...
1013;191;750;921;868;1658;703;...

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 `( emp_lname, ';' )`:

```
SELECT LIST( emp_id ORDER BY emp_lname, ';' ) AS "Sorted IDs"
FROM EMPLOYEE
GROUP BY dept_id
```

LOCATE function [String]

Function — 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.

Usage — 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 multi-byte characters are used, with the appropriate collation, then the starting position and the return value may be different from the *byte* positions.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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]

Function

Returns the natural logarithm of a number.

Syntax

LOG (*numeric-expression* **)**

Parameters

numeric-expression The number.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “LOG10 function [Numeric]” on page 178

Example — The following statement returns the value 3.912023.

```
SELECT LOG( 50 )
```

LOG10 function [Numeric]

Function — Returns the base 10 logarithm of a number.

Syntax — **LOG10 (** *numeric-expression* **)**

Parameters — **numeric-expression** The number.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “LOG function [Numeric]” on page 177

Example — The following statement returns the value 1.698970.

```
SELECT LOG10( 50 )
```

LONG_ULPLAN function [Miscellaneous]

Function — Returns a long description of the UltraLite plan optimization strategy of a SQL statement, as a string. The description is the same as that returned by the PLAN function.

For some queries, the execution plan for UltraLite may differ from the plan selected for Adaptive Server Anywhere.

Syntax — **LONG_ULPLAN (** *string-expression* **)**

Parameters — **string-expression** The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “PLAN function [Miscellaneous]” on page 198
- “EXPLANATION function [Miscellaneous]” on page 153
- “GRAPHICAL_PLAN function [Miscellaneous]” on page 157
- “GRAPHICAL_ULPLAN function [Miscellaneous]” on page 158
- “SHORT_ULPLAN function [Miscellaneous]” on page 220

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT LONG_ULPLAN(
   'select * from department where dept_id > 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 UltraLite Plan tab in the Results pane.

LOWER function [String]

Function

Converts all characters in a string to lower case.

Syntax

LOWER (*string-expression* **)**

Parameters

string-expression The string to be converted.

Standards and compatibility

- **SQL/92** SQL/92 compatible.
- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "LCASE function [String]" on page 172
- "UCASE function [String]" on page 238
- "UPPER function [String]" on page 238

Example

The following statement returns the value lower case.

```
SELECT LOWER( 'LOWER CasE' )
```

LTRIM function [String]

Function

Trims leading blanks from a string.

Syntax

LTRIM (*string-expression* **)**

Parameters

string-expression The string to be trimmed.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "RTRIM function [String]" on page 218
- "TRIM function [String]" on page 236

Example

The following statement returns the value Test Message with all leading blanks removed.

```
SELECT LTRIM( '     Test Message' )
```

MAX function [Aggregate]

Function

Returns the maximum *expression* value found in each group of rows.

Syntax

MAX (*expression* | **DISTINCT** *column name* **)**

Parameters

expression The expression for which the maximum value is to be calculated. This is commonly a column name.

DISTINCT column-name Returns the same as MAX(*expression*), and is included for completeness.

Usage

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Standards and compatibility

- ♦ **SQL/92** SQL/92 compatible.
- ♦ **SQL/99** Core feature.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

See also

- ♦ "MIN function [Aggregate]" on page 180

Example

The following statement returns the value 138948.000, representing the maximum salary in the employee table.

```
SELECT MAX( salary )
FROM employee
```

MIN function [Aggregate]

Function

Returns the minimum *expression* value found in each group of rows.

Syntax

MIN (*expression*
| **DISTINCT** *column name* **)**

Parameters

expression The expression for which the minimum value is to be calculated. This is commonly a column name.

DISTINCT column-name Returns the same as MIN(*expression*), and is included for completeness.

Usage

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Standards and compatibility

- ♦ **SQL/92** SQL/92 compatible.

- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "MAX function [Aggregate]" on page 180

Example

The following statement returns the value 24903.000, representing the minimum salary in the employee table.

```
SELECT MIN( salary )
FROM employee
```

MINUTE function [Date and time]

Function

Returns a number from 0 to 59 corresponding to the minute component of a datetime value.

Syntax

MINUTE (*datetime-expression* **)**

Parameters

datetime-expression The datetime value.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 22.

```
SELECT MINUTE( '1998-07-13 12:22:34' )
```

MINUTES function [Date and time]

Function

Given two timestamps, this function returns the integer number of minutes between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 131 instead for this purpose.

Given a single date, this function returns the number of minutes since 0000-02-29 00:00:00.

Given one date and an integer, it adds the integer number of minutes to the specified timestamp. Instead, please use the "DATEADD function [Date and time]" on page 131.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

Syntax 1

MINUTES ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

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.

☞ For information on casting data types, see "CAST function [Data type conversion]" on page 112.

Usage

Since this function returns an integer, overflow may occur when syntax 1 is used with timestamps greater than or equal to 4083-03-23 02:08:00.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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 1 051 040 527.

```
SELECT MINUTES( '1998-07-13 06:07:12' )
```

The following statements return the timestamp 1999-05-12 21:10:07.0. 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]

Function

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.

Usage

Division involving a negative dividend will give a negative or zero result. The sign of the divisor has no effect.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise. The % operator is used as a modulo operator in Adaptive Server Enterprise.

See also

- "REMAINDER function [Numeric]" on page 212

Example

The following statement returns the value 2.

```
SELECT MOD( 5, 3 )
```

MONTH function [Date and time]

Function

Returns a number from 1 to 12 corresponding to the month of the given date.

Syntax

MONTH (*date-expression* **)**

Parameters

date-expression A datetime value.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 7.

```
SELECT MONTH( '1998-07-13' )
```

MONTHNAME function [Date and time]

Function

Returns the name of the month from a date.

Syntax

MONTHNAME (*date-expression* **)**

Parameters

date-expression The datetime value.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value September.

```
SELECT MONTHNAME( '1998-09-05' )
```

MONTHS function [Date and time]

Function

Given two dates, this function returns the integer number of months between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 131 instead for this purpose.

Given a single date, this function returns the number of months since 0000-02.

Given one date and an integer, it adds the integer number of months to the specified date. It is recommended that you use the "DATEADD function [Date and time]" on page 131 instead for this purpose.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

Syntax 1

MONTHS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

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 on casting data types, see "CAST function [Data type conversion]" on page 112.

Usage

The value of MONTHS is calculated from the number of first days of the month between the two dates.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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 23 982.

```
SELECT MONTHS( '1998-07-13 06:07:12' )
```

The following statements return the timestamp 1999-10-12 21:05:07.0. 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' )
```

NEWID function [Miscellaneous]

Function

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 NEWID().

Usage

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 replication and synchronization environments.

NEWID is a non-deterministic function. Successive calls to NEWID 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" [*ASA SQL User's Guide*, page 449].

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "The NEWID default" [*ASA SQL User's Guide*, page 86]
- "STRTOUUID function [String]" on page 230
- "UUIDTOSTR function [String]" on page 239
- "UNIQUEIDENTIFIER data type [Binary]" on page 75

Example

The following statement creates a table 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 T1(
   pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
   c1 INT )
```

The following statement returns the unique identifier as a string:

```
SELECT NEWID()
```

For example, the value might be 0xd3749fe09cf446e399913bc6434f1f08.

NEXT_CONNECTION function [System]

Function

Returns an identifying number for a 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 first 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.

Usage

NEXT_CONNECTION can be used to enumerate the connections to a database. To get the first connection pass NULL; to get each subsequent connection, pass the previous return value. The function returns NULL when there are no more connections. The connection IDs are not returned in a particular order, but you can tell the order in which connections were made to the server using the connection ID. The first connection to a server is assigned the value 1, and for subsequent connections to the server, the connection IDs are incremented by a value of 1.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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 connection after *connection-id* on the current database.

```
NEXT_CONNECTION( connection-id )
```

The following call returns the connection after *connection-id* (regardless of database).

```
NEXT_CONNECTION( connection-id, NULL )
```

The following call returns the connection after *connection-id* on the specified database.

```
NEXT_CONNECTION( connection-id, database-id )
```

The following call returns the first connection (regardless of database).

```
NEXT_CONNECTION(NULL, NULL)
```

The following call returns the first connection on the specified database.

```
NEXT_CONNECTION( NULL, database-id )
```

NEXT_DATABASE function [System]

Function

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.

Usage

NEXT_DATABASE can be 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/92** Transact-SQL extension.
- ♦ **SQL/99** Transact-SQL extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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]

Function

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.

Usage

This function iterates over the HTTP headers included within a request. 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.

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 166
- "HTTP_VARIABLE function [HTTP]" on page 167
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 188

NEXT_HTTP_VARIABLE function [HTTP]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Usage

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.

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 166

- “HTTP_VARIABLE function [HTTP]” on page 167
- “NEXT_HTTP_HEADER function [HTTP]” on page 188

NOW function [Date and time]

Function

Returns the current year, month, day, hour, minute, second and fraction of a second. The fraction of a second. The accuracy is limited by the accuracy of the system clock.

The information the NOW function returns is equivalent to the information returned by the GETDATE function and the CURRENT TIMESTAMP special value.

Syntax

NOW (*)

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “GETDATE function [Date and time]” on page 157
- “CURRENT TIMESTAMP special value” on page 34

Example

The following statement returns the current date and time.

```
SELECT NOW(*)
```

NULLIF function [Miscellaneous]

Function

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.

Usage

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.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.

- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "CASE expressions" on page 18

Example

The following statement returns the value a:

```
SELECT NULLIF( 'a', 'b' )
```

The following statement returns NULL.

```
SELECT NULLIF( 'a', 'a' )
```

NUMBER function [Miscellaneous]

Function

Generates numbers starting at 1 for each successive row in the results of the query. NUMBER is primarily intended for use in select lists.

Syntax

NUMBER (*)

Usage

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 NUMBER 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 NUMBER in a DELETE statement, WHERE clause, HAVING clause, ORDER BY clause, subquery, query involving aggregation, any constraint, GROUP BY, DISTINCT, a query containing UNION ALL, 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(*)`.

NUMBER can also be used to generate primary keys when using the INSERT from SELECT statement (see "INSERT statement" on page 528), although using AUTOINCREMENT is a preferred mechanism for generating sequential primary keys.

☞ For information on AUTOINCREMENT, see "CREATE TABLE statement" on page 407.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Behavior changes
The behavior of the NUMBER function changed in version 8. For more information, see "Adaptive Server Anywhere behavior changes" [*What's New in SQL Anywhere Studio*, page 157].

- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns a sequentially-numbered list of departments.

```
SELECT NUMBER( * ), dept_name
FROM department
WHERE dept_id > 5
ORDER BY dept_name
```

OPENXML function [String]

Function

Generates a result set from an XML document.

Syntax

OPENXML (*xml-expression*,
xpath-query [, *flags* [, *namespace-declaration*]] **)**
WITH (*column-name column-type* [*xpath-query*],... **)**

Parameters

xml-expression The XML on which the result set is based. This can be any string expression, such as a constant, variable, or column.

xpath-query 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-query* argument generates one row in the table.

Metaproperties can only be specified in WITH clause *xpath-query* arguments. A metaproperty is accessed within an XPath query as if it was an attribute. If a *namespace-declaration* is not specified, then by default the prefix **mp** is bound to the Uniform Resource Identifier (URI) **urn:ianywhere-com:asa-xpath-metaprop**. If a *namespace-declaration* is specified, this URI must be bound to mp or some other prefix in order to access metaproperties in the query. OPENXML 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-query* 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-query* arguments are matched relative to the matches for the *xpath-query* 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 491.

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.

☞ For a list of data types, see "SQL Data Types" on page 53.

Usage

OPENXML parses the *xml-expression* 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 OPENXML are used to select nodes from the tree, and the selected nodes are then mapped to the result set.

The XML parser used by OPENXML 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 OPENXML.

☞ For information about the FOR XML clause and encoding binary data, see "FOR XML and binary data" [*ASA SQL User's Guide*, page 525].

OPENXML supports a subset of the XPath syntax, as follows:

- ♦ The child, self, attribute, descendant, descendant-or-self, and parent axes are fully supported.

  ☞ For descriptions, see "Using XPath expressions" [*ASA SQL User's Guide*, page 517].

- ♦ 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`.

☞ For information about the XPath query language, see *http://www.w3.org/TR/xpath*.

Standards and compatibility

- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Importing XML using OPENXML" [*ASA SQL User's Guide*, page 517]

Example

The following query generates a result set from the XML document supplied as the first argument to OPENXML:

```
SELECT * FROM OPENXML( '<products>
                 <prod_type id="301">Tee Shirt</prod_type>
                 <prod_type id="401">Baseball Cap</prod_type>
                 </products>',
                 '/products/prod_type' )
WITH ( prod_name LONG VARCHAR 'text()', prod_id CHAR(3) '@id')
```

This query generates the following result:

prod_name	prod_id
Tee Shirt	301
Baseball Cap	401

The following query uses an equality predicate to generate a result set from the supplied XML document.

```
SELECT * FROM OPENXML ('<emp_directory>
   <employee>
      <column name="emp_id">105</column>
      <column name="emp_fname">Matthew</column>
      <column name="emp_lname">Cobb</column>
      <column name="street">7 Pleasant Street</column>
      <column name="city">Grimsby</column>
      <column name="state">UT</column>
      <column name="zip_code">02154</column>
      <column name="phone">6175553840</column>
   </employee>
```

```
      <employee>
         <column name="emp_id">148</column>
         <column name="emp_fname">Julie</column>
         <column name="emp_lname">Jordan</column>
         <column name="street">1244 Great Plain Avenue</column>
         <column name="city">Woodbridge</column>
         <column name="state">AZ</column>
         <column name="zip_code">01890</column>
         <column name="phone">6175557835</column>
      </employee>
      <employee>
         <column name="emp_id">160</column>
         <column name="emp_fname">Robert</column>
         <column name="emp_lname">Breault</column>
         <column name="street">358 Cherry Street</column>
         <column name="city">Milton</column>
         <column name="state">PA</column>
         <column name="zip_code">02186</column>
         <column name="phone">6175553099</column>
      </employee>
      <employee>
         <column name="emp_id">243</column>
         <column name="emp_fname">Natasha</column>
         <column name="emp_lname">Shishov</column>
         <column name="street">151 Milk Street</column>
         <column name="city">Grimsby</column>
         <column name="state">UT</column>
         <column name="zip_code">02154</column>
         <column name="phone">6175552755</column>
      </employee>
   </emp_directory>', '/emp_directory/employee')
   WITH ( emp_id INT 'column[@name="emp_id"]',
          first_name           CHAR(20) 'column[@name="emp_
            fname"]',
          last_name              CHAR(20) 'column[@name="emp_
            lname"]',
          phone_number      CHAR(10) 'column[@name="phone"]')
```

This query generates the following result set:

emp_id	first_name	last_name	phone_number
105	Matthew	Cobb	6175553840
148	Julie	Jordan	6175557835
160	Robert	Breault	6175553099
243	Natasha	Shishov	6175552755

☞ For more examples of using OPENXML, see "Importing XML using OPENXML" [*ASA SQL User's Guide*, page 517].

PATINDEX function [String]

Function

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, PATINDEX 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.

Usage

PATINDEX returns the starting position of the first occurrence of the pattern. If the pattern is not found, it returns zero (0).

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise, except that the USING clause is not supported.

See also

- "LIKE conditions" on page 25
- "LOCATE function [String]" on page 176

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 [Aggregate]

Function

Computes the (fractional) position of one row returned from a query with respect to the other rows returned by the query, as defined by the

ORDER BY clause. It returns a decimal value between 0 and 1.

Syntax

PERCENT_RANK ( )

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/OLAP feature T611
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CUME_DIST function [Aggregate]" on page 129
- "DENSE_RANK function [Aggregate]" on page 142
- "RANK function [Aggregate]" on page 203
- "ROW_NUMBER function [Aggregate]" on page 217

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 dept_id, emp_lname, salary, sex,
PERCENT_RANK () OVER (PARTITION BY sex
ORDER BY salary DESC) "Rank"
FROM employee
WHERE state IN ('NY')
```

dept_id	emp_lname	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]

Function

Returns the numeric value PI.

Syntax

PI (*)

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** The PI() function is supported in Adaptive Server Enterprise, but PI(*) is not.

Example

The following statement returns the value 3.141592653. . .

```
SELECT PI( * )
```

PLAN function [Miscellaneous]

Function

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.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "EXPLANATION function [Miscellaneous]" on page 153
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 157
- "GRAPHICAL_ULPLAN function [Miscellaneous]" on page 158
- "LONG_ULPLAN function [Miscellaneous]" on page 178
- "SHORT_ULPLAN function [Miscellaneous]" on page 220

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT PLAN(
   'SELECT * FROM department WHERE dept_id > 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 department where ...'`.

```
SELECT PLAN(
   'SELECT * FROM department WHERE dept_id > 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]

Function

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value 64.

```
SELECT Power( 2, 6 )
```

PROPERTY_DESCRIPTION function [System]

Function

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.

Usage

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Database Performance and Connection Properties" [*ASA Database Administration Guide*, page 703]

Example

The following statement returns the description Number of index insertions.

```
SELECT PROPERTY_DESCRIPTION( 'IndAdd' )
```

PROPERTY function [System]

Function

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.

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Server-level properties" [*ASA Database Administration Guide*, page 725]

Example

The following statement returns the name of the current database server:

```
SELECT PROPERTY( 'Name' )
```

PROPERTY_NAME function [System]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Database properties" [*ASA Database Administration Guide*, page 713]

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]

Function

Returns the property number of the property with the supplied property-name.

Syntax

PROPERTY_NUMBER (*property-name* **)**

Parameters

property-name A property name.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Database properties" [*ASA Database Administration Guide*, page 713]

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]

Function

Returns a number indicating the quarter of the year from the supplied date expression.

Syntax

QUARTER (*date-expression* **)**

Parameters

date- expression The date.

Usage

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/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 2.

```
SELECT QUARTER ( '1987/05/02' )
```

RADIANS function [Numeric]

Function

Converts a number from degrees to radians.

Syntax

RADIANS (*numeric-expression* **)**

Parameters

numeric-expression A number, in degrees. This angle is converted to radians.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns a value of approximately 0.5236.

```
SELECT RADIANS( 30 )
```

RAND function [Numeric]

Function

Returns a random number in the interval 0 to 1, with an optional seed.

Syntax

RAND ([*integer-expression*] **)**

Parameters

integer-expression The optional seed used to create a random number. This argument allows you to create repeatable random number sequences.

Usage

RAND is a non-deterministic function. Successive calls to RAND may return different values. The query optimizer does not cache the results of the RAND function.

☞ For more information about non-deterministic functions, see "Function caching" [*ASA SQL User's Guide*, page 449].

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns a value of approximately 0.0554504.

```
SELECT RAND( 4 )
```

RANK function [Aggregate]

Function

Calculates the value of a rank in a group of values. In the case of ties, RANK leaves a gap in the ranking sequence.

Syntax

RANK ( )

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/OLAP feature T611
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CUME_DIST function [Aggregate]" on page 129
- "DENSE_RANK function [Aggregate]" on page 142
- "PERCENT_RANK function [Aggregate]" on page 196
- "ROW_NUMBER function [Aggregate]" on page 217

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 emp_lname, salary, state,
RANK () OVER (ORDER BY salary DESC) "Rank"
FROM employee WHERE state IN ('NY','UT')
```

emp_lname	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	7
Driscoll	48023.690	UT	9

emp_lname	salary	state	Rank
Hildebrand	45829.000	UT	10
Whitney	45700.000	NY	11
...	...	...	...
Lynch	24903.000	UT	19

REGR_AVGX [Aggregate]

Function

Computes the average of the independent variable of the regression line.

Syntax

REGR_AVGX (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

AVG (*y*)

where *y* represents the *independent-expression.*

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "AVG function [Aggregate]" on page 109
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGY [Aggregate]" on page 205

Example

The following example calculates the average of the dependent variable, employee age. The function returns the value 45.44.

```
SELECT REGR_AVGX (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_AVGY [Aggregate]

Function

Computes the average of the dependent variable of the regression line.

Syntax

REGR_AVGY (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

AVG (*x*)

where *x* represents the *dependent-expression.*

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGX [Aggregate]" on page 204
- "AVG function [Aggregate]" on page 109

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(birth_date)))
FROM employee
```

REGR_COUNT [Aggregate]

Function Returns an integer that represents the number of non-null number pairs used to fit the regression line.

Syntax **REGR_COUNT (** *dependent-expression* , *independent-expression* **)**

Parameters **dependent-expression** The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGY [Aggregate]" on page 205
- "REGR_AVGX [Aggregate]" on page 204
- "COUNT function [Aggregate]" on page 125
- "AVG function [Aggregate]" on page 109
- "SUM function [Aggregate]" on page 233

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(birth_date)))
FROM employee
```

REGR_INTERCEPT [Aggregate]

Function Computes the y-intercept of the regression line.

Syntax **REGR_INTERCEPT (** *dependent-expression* , *independent-expression* **)**

Parameters **dependent-expression** The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

AVG (*x*) - **REGR_SLOPE** (*x*, *y*) * **AVG** (*y*)

where *x* represents the *dependent-expression* and *y* represents the *independent-expression*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_COUNT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGY [Aggregate]" on page 205
- "REGR_AVGX [Aggregate]" on page 204
- "REGR_SLOPE [Aggregate]" on page 208
- "AVG function [Aggregate]" on page 109

Example

The following example returns the value 7,486.6384185191855.

```
SELECT REGR_INTERCEPT (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_R2 [Aggregate]

Function

Computes the coefficient of determination (also referred to as *R-squared* or the*goodness of fit*) for the regression line.

Syntax

REGR_R2 (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGX [Aggregate]" on page 204
- "REGR_AVGY [Aggregate]" on page 205

Example

The following example returns the value 0.19379959710325653.

```
SELECT REGR_R2 (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_SLOPE [Aggregate]

Function

Computes the slope of the regression line fitted to non-null pairs.

Syntax

REGR_SLOPE (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

COVAR_POP (x , y) / **VAR_POP** (y)

where x represents the *dependent-expression* and y represents the *independent-expression.*

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210
- "REGR_SYY [Aggregate]" on page 211
- "REGR_AVGX [Aggregate]" on page 204
- "REGR_AVGY [Aggregate]" on page 205
- "COVAR_POP function [Aggregate]" on page 126
- "VAR_POP function [Aggregate]" on page 240

Example

The following example returns the value 935.3429749445614.

```
SELECT REGR_SLOPE (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_SXX [Aggregate]

Function

Returns values that can evaluate the statistical validity of a regression model.

Syntax

REGR_SXX (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

REGR_COUNT (*x* , *y*) ***** **VAR_POP** (*y*)

where *x* represents the *dependent-expression* and *y* represents the *independent-expression.*

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “REGR_COUNT [Aggregate]” on page 206
- “REGR_INTERCEPT [Aggregate]” on page 206
- “REGR_COUNT [Aggregate]” on page 206
- “REGR_AVGX [Aggregate]” on page 204
- “REGR_AVGY [Aggregate]” on page 205
- “REGR_SXY [Aggregate]” on page 210
- “REGR_SYY [Aggregate]” on page 211
- “VAR_POP function [Aggregate]” on page 240

Example

The following example returns the value 5916.4800000000105.

```
SELECT REGR_SXX (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_SXY [Aggregate]

Function

Returns values that can evaluate the statistical validity of a regression model.

Syntax

REGR_SXY (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

REGR_COUNT (*x* , *y*) * **VAR_POP** (*x*)

where *x* represents the *dependent-expression* and *y* represents the *independent-expression.*

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “REGR_COUNT [Aggregate]” on page 206
- “REGR_INTERCEPT [Aggregate]” on page 206
- “REGR_COUNT [Aggregate]” on page 206
- “REGR_SLOPE [Aggregate]” on page 208

- "REGR_AVGX [Aggregate]" on page 204
- "REGR_AVGY [Aggregate]" on page 205
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SYY [Aggregate]" on page 211

Example

The following example returns the value 5533938.004400015.

```
SELECT REGR_SXY (salary, (year(now()) - year(birth_date)))
FROM employee
```

REGR_SYY [Aggregate]

Function

Returns values that can evaluate the statistical validity of a regression model.

Syntax

REGR_SYY (*dependent-expression* , *independent-expression* **)**

Parameters

dependent-expression The variable that is affected by the independent variable.

independent-expression The variable that influences the outcome.

Usage

The function returns a numeric value. 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:

REGR_COUNT (*x* , *y*) * **COVAR_POP** (*x*, *y*)

where *x* represents the *dependent-expression* and *y* represents the *independent-expression*.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "REGR_COUNT [Aggregate]" on page 206
- "REGR_INTERCEPT [Aggregate]" on page 206
- "REGR_COUNT [Aggregate]" on page 206
- "REGR_AVGX [Aggregate]" on page 204
- "REGR_AVGY [Aggregate]" on page 205
- "REGR_SLOPE [Aggregate]" on page 208
- "REGR_SXX [Aggregate]" on page 209
- "REGR_SXY [Aggregate]" on page 210

Example

The following example returns the value 26, 708, 672,843.3002.

```
SELECT REGR_SYY (salary, (year(now()) - year(birth_date)))
FROM employee
```

REMAINDER function [Numeric]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. The % (modulo) operator and the division operator can be used to produce a remainder.

See also

- "MOD function [Numeric]" on page 182

Example

The following statement returns the value 2.

```
SELECT REMAINDER( 5, 3 )
```

REPEAT function [String]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise, but REPLICATE provides the same capabilities.

See also

- "REPLICATE function [String]" on page 213

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPEAT( 'repeat', 3 )
```

REPLACE function [String]

Function

Replaces all occurrences of a substring with another substring.

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "SUBSTRING function [String]" on page 232

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 would need to 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.SYSTABLE
WHERE table_id <= 5
```

REPLICATE function [String]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "REPEAT function [String]" on page 212

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPLICATE( 'repeat', 3 )
```

REWRITE function [Miscellaneous]

Function

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.

Usage

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 Adaptive Server 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Semantic query transformations" [*ASA SQL User's Guide*, page 435]
- "EXTENDED_JOIN_SYNTAX option [database]" [*ASA Database Administration Guide*, page 653]
- "Transact-SQL outer joins (*= or =*)" [*ASA SQL User's Guide*, page 280]
- "Key joins" [*ASA SQL User's Guide*, page 294]
- "Natural joins" [*ASA SQL User's Guide*, page 290]
- "Duplicate correlation names in joins (star joins)" [*ASA SQL User's Guide*, page 284]

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 employee and sales_order tables. The second optimization simplifies the query by eliminating the primary key - foreign key join between employee and sales_order. Part of this rewrite optimization is to replace the join predicate e.emp_id=s.sales_rep with the predicate s.sales_rep IS NOT NULL.

```
SELECT REWRITE( 'SELECT s.id, s.order_date
    FROM sales_order s
    WHERE  EXISTS(SELECT *
       FROM employee e
          WHERE e.emp_id = s.sales_rep)' ) FROM dummy
```

The query returns a single column result set containing the rewritten query:

```
'SELECT s.id, s.order_date FROM sales_order s WHERE s.sales_rep
         IS NOT NULL'
```

The next example of REWRITE uses the ANSI argument.

```
SELECT REWRITE( 'SELECT DISTINCT s.id, s.order_date, e.emp_
         fname, e.emp_id
    FROM sales_order s, employee e
          WHERE e.emp_id *= s.sales_rep', '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.order_date, e.emp_id, e.emp_fname
 FROM employee as e
 LEFT OUTER JOIN  sales_order as s
 ON e.emp_id = s.sales_rep’
```

RIGHT function [String]

Function	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.
Usage	If the string contains multi-byte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Compatible with Adaptive Server Enterprise.
See also	♦ “LEFT function [String]” on page 172 ♦ “International Languages and Character Sets” [*ASA Database Administration Guide*, page 319]
Example	The following statement returns the value olate.

```
SELECT RIGHT( ’chocolate’, 5 )
```

ROUND function [Numeric]

Function	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.
Standards and compatibility	♦ **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "TRUNCNUM function [Numeric]" on page 237

Example

The following statement returns the value 123.200.

```
SELECT ROUND( 123.234, 1 )
```

ROW_NUMBER function [Aggregate]

Function

Assigns a unique number to each row.

Syntax

ROW_NUMBER ( )

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/OLAP feature T611
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "CUME_DIST function [Aggregate]" on page 129
- "DENSE_RANK function [Aggregate]" on page 142
- "PERCENT_RANK function [Aggregate]" on page 196
- "RANK function [Aggregate]" on page 203

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 emp_lname, salary, state,
ROW_NUMBER () OVER (ORDER BY salary DESC) "Rank"
FROM employee WHERE state IN ('NY','UT')
```

emp_lname	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

emp_lname	salary	state	Rank
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]

Function
Returns a string with trailing blanks removed.

Syntax
RTRIM (*string-expression* **)**

Parameters
string-expression The string to be trimmed.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "LTRIM function [String]" on page 179

Example
The following statement returns the string Test Message, with all trailing blanks removed.

```
SELECT RTRIM( 'Test Message     ' )
```

SECOND function [Date and time]

Function
Returns a number from 0 to 59 corresponding to the second component of the given datetime value.

Syntax
SECOND (*datetime-expression* **)**

Parameters
datetime-expression The datetime value.

Standards and compatibility

- **SQL/92** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example
The following statement returns the value 21.

```
SELECT SECOND( '1998-07-13:21:21:25' )
```

SECONDS function [Date and time]

Function

Given two timestamps, this function returns the integer number of seconds between them. It is recommended that you use the "DATEDIFF function [Date and time]" on page 131 instead for this purpose.

Given a single date, this function returns the number of seconds since 0000-02-29 00:00:00.

Given one date and an integer, it adds the integer number of seconds to the specified timestamp. It is recommended that you use the "DATEADD function [Date and time]" on page 131 instead for this purpose.

Syntax 1 returns a bigint. Syntax 2 returns a timestamp.

Syntax 1

SECONDS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

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.

☞ For information on casting data types, see "CAST function [Data type conversion]" on page 112.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statements return the value 14 400, signifying that the second timestamp is 14 400 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 63 062 431 632.

```
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, '1999-05-12 21:05:07' )
```

SHORT_ULPLAN function [Miscellaneous]

Function

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.

For some queries, the execution plan for UltraLite may differ from the plan selected for Adaptive Server Anywhere.

Syntax

SHORT_ULPLAN (*string-expression* **)**

Parameters

string-expression The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "PLAN function [Miscellaneous]" on page 198
- "EXPLANATION function [Miscellaneous]" on page 153
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 157
- "GRAPHICAL_ULPLAN function [Miscellaneous]" on page 158
- "LONG_ULPLAN function [Miscellaneous]" on page 178

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT EXPLANATION(
   'select * from department where dept_id > 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 UltraLite Plan tab in the Results pane.

SIGN function [Numeric]

Function

Returns the sign of a number.

Syntax

SIGN (*numeric-expression* **)**

Parameters

numeric-expression The number for which the sign is to be returned.

Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns the value -1

```
SELECT SIGN( -550 )
```

Return value

For negative numbers, SIGN returns -1.

For zero, SIGN returns 0.

For positive numbers, SIGN returns 1.

SIMILAR function [String]

Function

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.

Usage

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.

Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value 75.

```
SELECT SIMILAR( 'toast', 'coast' )
```

This signifies that the two values are 75% similar.

SIN function [Numeric]

Function

Returns the sine of a number.

Syntax

SIN (*numeric-expression* **)**

Parameters

numeric-expression The angle, in radians.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "ASIN function [Numeric]" on page 108
- "COS function [Numeric]" on page 124
- "COT function [Numeric]" on page 124
- "TAN function [Numeric]" on page 234

Example

The following statement returns the value 0.496880.

```
SELECT SIN( 0.52 )
```

SORTKEY function [String]

Function

Generates values that can be used to sort character strings based on alternate collation rules.

Syntax

SORTKEY (*string-expression*
[, *collation-name* | , *collation-id*] **)**

Parameters

string-expression The string expression may only contain characters that are encoded in the database's character set.

If *string-expression* is an empty string, SORTKEY returns a zero-length binary value. If *string-expression* is null, SORTKEY 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 SORTKEY 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.

collation-id A variable, integer constant, or string that specifies the ID number of the sort order to use.

If you do not specify a collation, the default is Default Unicode multilingual.

Following are the valid values for *collation-name* and *collation-id*:

Description	Collation name	Collation ID
Default Unicode multilingual	default	0
CP 850 Alternative: no accent	altnoacc	39
CP 850 Alternative: lower case 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
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

Description	Collation name	Collation ID
ISO 8859-5 Turkish no accents	turnoac	73
ISO 8859-5 Turkish no case	turnocs	74
CP 874 (TIS 620) Royal Thai dictionary	thaidict	257
ISO 14651 ordering standard	14651	258
Shift-JIS binary order	sjisbin	259
Unicode UTF-8 binary sort	utf8bin	260
EUC JIS binary order	eucjisbin	261
GB2312 binary order	gb2312bn	262
CP932 MS binary order	cp932bin	263
Big5 binary order	big5bin	264
EUC KSC binary order	eucksсbin	265

Usage

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 are beyond the limitation of Adaptive Server Anywhere collations. The returned value is a binary value that contains coded sort order information for the input string is retained from the SORTKEY function.

For example, you can store the values returned by SORTKEY 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 SORTKEY.

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.

The input of SORTKEY can generate up to six bytes of sort order information for each input character. The output of SORTKEY is of type VARBINARY and has a maximum length of (254 * 6) bytes.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Compatible with Adaptive Server Enterprise, except that

Adaptive Server Enterprise does not allow the use of self-defined sort orders.

See also

- ♦ "COMPARE function [String]" on page 115
- ♦ "Introduction to international languages and character sets" [*ASA Database Administration Guide*, page 320]

Example

The following statements return the sort key values in the sort order: Latin-1, English, French, German dictionary.

```
SELECT SORTKEY('coop', 'dict')
```

SORTKEY('coop', 'dict')
0x08890997099709b30008000800080008

```
SELECT SORTKEY ( 'Cö-op', 51 )
```

SORTKEY('Cö-op', 51)
0x08890997099709b30020004700020008000800080001fffd002d

SOUNDEX function [String]

Function

Returns a number representing the sound of a string.

Syntax

SOUNDEX (*string-expression* **)**

Parameters

string-expression The string.

Usage

The SOUNDEX function value for a string is based on the first letter and the next three consonants other than H, Y, and W. Doubled letters are counted as one letter. For example,

```
SOUNDEX( 'apples' )
```

is based on the letters A, P, L, and S.

Multi-byte characters are ignored by the SOUNDEX function.

Although it is not perfect, SOUNDEX will normally return 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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.

- **Sybase** Compatible with Adaptive Server Enterprise, except that Adaptive Server Enterprise returns a CHAR(4) result and Adaptive Server Anywhere returns an integer.

Example

The following statement returns two numbers, representing the sound of each name. The SOUNDEX value for each argument is 3827.

```
SELECT SOUNDEX( 'Smith' ), SOUNDEX( 'Smythe' )
```

SPACE function [String]

Function

Returns a specified number of spaces.

Syntax

SPACE (*integer-expression* **)**

Parameters

integer-expression The number of spaces to return.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

The following statement returns a string containing 10 spaces.

```
SELECT SPACE( 10 )
```

SQLDIALECT function [Miscellaneous]

Function

Returns either ‘Watcom-SQL’ or ‘Transact-SQL’, indicating 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- “TRANSACTSQL function [Miscellaneous]” on page 235
- “WATCOMSQL function [Miscellaneous]” on page 242

Example

The following statement returns the string Transact-SQL.

```
SELECT
   SQLDIALECT( 'SELECT employeeName = emp_lname FROM employee' )
FROM dummy
```

SQRT function [Numeric]

Function
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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example
The following statement returns the value 3.

```
SELECT SQRT( 9 )
```

STDDEV function [Aggregate]

Function
An alias for STDDEV_SAMP. See "STDDEV_SAMP function [Aggregate]" on page 228.

STDDEV_POP function [Aggregate]

Function
Computes the standard deviation of a population consisting of a numeric-expression, as a DOUBLE.

Syntax
STDDEV_POP (*numeric-expression* **)**

Parameters
numeric-expression The expression whose population-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Usage
The population-based standard deviation (s) is computed according to the following formula:

$s = [\ (1/N) * \mathrm{SUM}(\ x_i - \mathrm{mean}(\ x\ )\ )^2\ ]^{1/2}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Aggregate functions" on page 92

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT year( ship_date ) AS Year,
    quarter( ship_date ) AS Quarter,
    avg( quantity ) AS Average,
    STDDEV_POP( quantity ) AS Variance
FROM sales_order_items
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]

Function

Computes the standard deviation of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax

STDDEV_SAMP (*numeric-expression* **)**

Parameters

numeric-expression The expression whose sample-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Usage

The standard deviation (s) is computed according to the following formula, which assumes a normal distribution:

$s = [(1/(N - 1)) * \mathrm{SUM}(x_i - \mathrm{mean}(x))^2]^{1/2}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** SQL/foundation feature outside of core SQL.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ “Aggregate functions” on page 92

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT year( ship_date ) AS Year,
    quarter( ship_date ) AS Quarter,
    avg( quantity ) AS Average,
    STDDEV_SAMP( quantity ) AS Variance
FROM sales_order_items
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]

Function

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.

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

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]

Function

Concatenates one or more strings into one large string.

Syntax

STRING (*string-expression* [, . . .] **)**

Parameters

string-expression A string.

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.

Usage

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement returns the value *testing123*.

```
SELECT STRING( 'testing', NULL, 123 )
```

STRTOUUID function [String]

Function

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 is defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions are 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 is a native data type and Adaptive Server Anywhere carries out conversions as needed. The STRTOUUID and UUIDTOSTR functions are therefore not needed.

☞ For more information, see "UNIQUEIDENTIFIER data type [Binary]" on page 75.

Syntax

STRTOUUID (*string-expression* **)**

Parameters

string-expression A string in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "UUIDTOSTR function [String]" on page 239
- "NEWID function [Miscellaneous]" on page 185

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]

Function

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 STUFF, use a replacement string of NULL.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "INSERTSTR function [String]" on page 169

Example

The following statement returns the value **chocolate pie**.

```
SELECT STUFF( 'chocolate cake', 11, 4, 'pie' )
```

SUBSTRING function [String]

Function

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. A negative starting position specifies a number of characters from the end of the string instead of the beginning. The first character in the string is at position 1.

length The length of the substring to return, in characters. A positive *length* specifies that the substring ends *length* characters to the right of the starting position, while a negative *length* returns at most *length* characters up to, and including, the starting position, from the left of the starting position.

Usage

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 *length* is specified, the substring 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 *string-expression* is of binary data type, the SUBSTRING function behaves as BYTE_SUBSTR.

Standards and compatibility

- **SQL/92** Entry-level feature.

- **SQL/99** Core feature.
- **Sybase** SUBSTRING is compatible with Adaptive Server Enterprise. SUBSTR is not supported by Adaptive Server Enterprise.

See also

- "BYTE_SUBSTR function [String]" on page 111

Example

The following statement returns **back**:

```
SELECT SUBSTRING( 'back yard', 1, 4 )
```

The following statement returns **yard**:

```
SELECT SUBSTRING( 'back yard', 6, 4 )
```

The following statement returns **gh**:

```
SELECT SUBSTR('abcdefgh',0,-2)
```

The following statement returns **gh**:

```
SELECT SUBSTR('abcdefgh',-2,2)
```

The following statements both return **ab**:

```
SELECT SUBSTR('abcdefgh',2,-2)
```

```
SELECT SUBSTR('abcdefgh',2,-4)
```

The following statement returns **b**:

```
SELECT SUBSTR('abcdefgh',2,-1)
```

SUM function [Aggregate]

Function

Returns the total of the specified expression for each group of rows.

Syntax

SUM (*expression* | **DISTINCT** *column-name* **)**

Parameters

expression The object to be summed. This is commonly a column name.

DISTINCT column-name This is of limited usefulness, but is included for completeness.

Usage

Rows where the specified expression is NULL are not included.

Returns NULL for a group containing no rows.

Standards and compatibility

- **SQL/92** SQL/92 compatible.
- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “COUNT function [Aggregate]” on page 125
- “AVG function [Aggregate]” on page 109

Example

The following statement returns the value 3749146.

```
SELECT SUM( salary )
FROM Employee
```

TAN function [Numeric]

Function

Returns the tangent of a number.

Syntax

TAN (*numeric-expression* **)**

Parameters

numeric-expression An angle, in radians.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “COS function [Numeric]” on page 124
- “SIN function [Numeric]” on page 221

Example

The following statement returns the value 0.572561.

```
SELECT TAN( 0.52 )
```

TEXTPTR function [Text and image]

Function

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.

Usage

This function is included for Transact-SQL compatibility.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

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
```

TODAY function [Date and time]

Function

Returns the current date. This is the historical syntax for CURRENT DATE.

Syntax

TODAY (*)

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statements return the current day according to the system clock.

```
SELECT TODAY( * )
SELECT CURRENT DATE
```

TRACEBACK function [Miscellaneous]

Function

Returns a string containing a traceback of the procedures and triggers that were executing when the most recent exception (error) occurred.

Syntax

TRACEBACK (*)

Usage

This is useful for debugging procedures and triggers

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

To use the traceback function, enter the following after an error occurs while executing a procedure:

```
SELECT TRACEBACK ( * )
```

TRANSACTSQL function [Miscellaneous]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "SQLDIALECT function [Miscellaneous]" on page 226
- "WATCOMSQL function [Miscellaneous]" on page 242

Example

The following statement returns the string `'select EmployeeName = empl_name from employee'`.

```
SELECT TRANSACTSQL( 'SELECT empl_name as EmployeeName FROM
         employee' ) FROM dummy
```

TRIM function [String]

Function

Removes leading and trailing blanks from a string.

Syntax

TRIM (*string-expression* **)**

Parameters

string-expression The string to be trimmed.

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "LTRIM function [String]" on page 179
- "RTRIM function [String]" on page 218

Example

The following statement returns the value *chocolate* with no leading or trailing blanks.

```
SELECT TRIM( '   chocolate   ' )
```

TRUNCATE function [Numeric]

Function

Truncates a number at a specified number of places after the decimal point. Deprecated in favor of TRUNCNUM.

Syntax

"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.

Usage

This function is the same as TRUNCNUM. Using TRUNCNUM is recommended as it does not cause keyword conflicts.

The quotation marks are required because of a keyword conflict with the TRUNCATE TABLE statement. You can only use TRUNCATE without the quotation marks if the QUOTED_IDENTIFIER option is set to OFF.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- "QUOTED_IDENTIFIER option [compatibility]" [*ASA Database Administration Guide*, page 685]
- "TRUNCNUM function [Numeric]" on page 237

Example

The following statement returns the value 600.

```
SELECT "TRUNCATE"( 655, -2 )
```

The following statement returns the value 655.340.

```
SELECT "TRUNCATE"( 655.348, 2 )
```

TRUNCNUM function [Numeric]

Function

Truncates a number at a specified number of places after the decimal point.

Syntax

TRUNCNUM (*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.

Usage

This function is the same as TRUNCATE, but does not cause keyword conflicts.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

See also

- "ROUND function [Numeric]" on page 216

- “TRUNCATE function [Numeric]” on page 236

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]

Function Converts all characters in a string to upper case.

Syntax **UCASE (** *string-expression* **)**

Parameters **string-expression** The string to be converted to upper case.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** UCASE is not supported by Adaptive Server Enterprise, but UPPER provides the same feature in a compatible manner.

See also

- “UPPER function [String]” on page 238
- “LCASE function [String]” on page 172

Example The following statement returns the value CHOCOLATE.

```
SELECT UCASE( 'ChocoLate' )
```

UPPER function [String]

Function Converts all characters in a string to upper case.

Syntax **UPPER (** *string-expression* **)**

Parameters **string-expression** The string to be converted to upper case.

Standards and compatibility

- **SQL/92** This function is SQL/92 compatible.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- “UCASE function [String]” on page 238
- “LCASE function [String]” on page 172
- “LOWER function [String]” on page 179

Example

The following statement returns the value CHOCOLATE.

```
SELECT UPPER( 'ChocoLate' )
```

UUIDTOSTR function [String]

Function

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 is defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions are 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 is a native data type and Adaptive Server Anywhere carries out conversions as needed. The STRTOUUID and UUIDTOSTR functions are therefore not needed.

☞ For more information, see "UNIQUEIDENTIFIER data type [Binary]" on page 75.

Syntax

UUIDTOSTR (*uuid-expression* **)**

Parameters

uuid-expression A unique identifier value.

Usage

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "NEWID function [Miscellaneous]" on page 185
- "STRTOUUID function [String]" on page 230

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 T1(
    pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    c1 INT );
INSERT INTO T1( c1 ) values ( 1 );
INSERT INTO T1( c1 ) values ( 2 );
```

Executing the following SELECT statement returns all of the data in the newly created table.

```
SELECT *
FROM T1
```

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 T1
```

The UUIDTOSTR function is not needed for database created with version 9.0.2 or later.

VAR_POP function [Aggregate]

Function

Computes the statistical variance of a population consisting of a numeric-expression, as a DOUBLE.

Syntax

VAR_POP (*numeric-expression* **)**

Parameters

numeric-expression The expression whose population-based variance is calculated over a set of rows. The expression is commonly a column name.

Usage

The population-based variance (s^2) is computed according to the following formula:

$s^2 = (1/N) * \mathrm{SUM}(x_i - \mathrm{mean}(x))^2$

This variance does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Aggregate functions" on page 92

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT year( ship_date ) AS Year,
    quarter( ship_date ) AS Quarter,
    avg( quantity ) AS Average,
    var_pop( quantity ) AS Variance
FROM sales_order_items
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]

Function

Computes the statistical variance of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax

VAR_SAMP (*numeric-expression* **)**

Parameters

numeric-expression The expression whose sample-based variance is calculated over a set of rows. The expression is commonly a column name.

Usage

The variance (s^2) 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 no rows.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL. The VARIANCE syntax is a vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Aggregate functions" on page 92

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT year( ship_date ) AS Year,
    quarter( ship_date ) AS Quarter,
    avg( quantity ) AS Average,
    var_samp( quantity ) AS Variance
FROM sales_order_items
GROUP BY Year, Quarter
ORDER BY Year, Quarter
```

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

VARIANCE function [Aggregate]

Function

An alias for VAR_SAMP. See "VAR_SAMP function [Aggregate]" on page 241.

VAREXISTS function [Miscellaneous]

Function

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 *name to be tested, as a* string.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise.

See also

- "CREATE VARIABLE statement" on page 427
- "DECLARE statement" on page 435
- "IF statement" on page 518

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]

Function

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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- ♦ "SQLDIALECT function [Miscellaneous]" on page 226
- ♦ "TRANSACTSQL function [Miscellaneous]" on page 235

Example

The following statement returns the string `'select empl_name as EmployeeName from employee'`.

```
SELECT WATCOMSQL( 'SELECT EmployeeName=empl_name FROM employee'
        ) FROM dummy
```

WEEKS function [Date and time]

Function

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 131 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 131 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 on casting data types, see "CAST function [Data type conversion]" on page 112.

Usage

The difference of two dates in weeks is the number of Sundays between the two dates.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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 104 270.

```
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, '1999-05-12 21:05:07' )
```

XMLAGG function [String]

Function

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.

Usage

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" [*ASA SQL User's Guide*, page 544].

Standards and compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Using the XMLAGG function" [*ASA SQL User's Guide*, page 544]

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 "product",
                           XMLATTRIBUTES( prod_id,
                              quantity
                              AS "quantity_shipped" )
                                        )
                                    )
                      FROM sales_order_items soi
                      WHERE soi.id = so.id
                    )
                  ) AS products_ordered
FROM sales_order so
ORDER BY so.id
```

XMLCONCAT function [String]

Function

Produces a forest of XML elements.

Syntax

XMLCONCAT (*xml-value*,... **)**

Parameters

xml-value The XML values to be concatenated.

Usage

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. XMLCONCAT 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 XMLCONCAT.

Standards and Compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Using the XMLCONCAT function" [*ASA SQL User's Guide*, page 545]
- "XMLFOREST function [String]" on page 247

Example

The following query generates <cust_id>, <cust_fname>, and <cust_lname> elements for each customer.

```
SELECT XMLCONCAT( XMLELEMENT ( NAME cust_id, id ),
                  XMLELEMENT( NAME cust_fname, fname ),
                  XMLELEMENT( NAME cust_lname, lname )
                 ) AS "Customer Information"
FROM customer
WHERE id < 120
```

XMLELEMENT function [String]

Function

Produces an XML element within a query.

Syntax

XMLELEMENT (NAME *element-name-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,

```
SELECT XMLELEMENT( NAME x, 'abc', 'def' )
```

returns

```
<x>abcdef</x>
```

Usage

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 more information about quoting and the XMLELEMENT function, see "Invalid names and SQL/XML" [*ASA SQL User's Guide,* page 544].

XMLELEMENT functions can be nested to create a hierarchy. If you wish 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 247.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains XMLELEMENT.

Standards and compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Using the XMLELEMENT function" [*ASA SQL User's Guide*, page 546]
- "XMLFOREST function [String]" on page 247

Example

The following statement 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 product p
WHERE id > 400
```

XMLFOREST function [String]

Function

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 emp_id column from the employee table for this argument, then an <emp_id> element containing an emp_id value is generated for each value in the table.

Specify the *element-name* argument if you wish to assign a name other than the *element-content-expression* to the element, otherwise the element name defaults to the *element-content-expression* name.

Usage

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 XMLFOREST 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 XMLFOREST. Use the XMLELEMENT function if you wish to specify attributes for generated elements.

☞ For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" on page 246.

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.

Standards and compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Using the XMLFOREST function" [*ASA SQL User's Guide*, page 548]
- "XMLELEMENT function [String]" on page 246
- "XMLCONCAT function [String]" on page 245

Example

The following statement produces an XML element for the first and last name of each employee.

```
SELECT emp_id,
       XMLFOREST( emp_fname, emp_lname
                ) AS "Employee Name"
FROM employee
```

XMLGEN function [String]

Function

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 emp_id="{$emp_id}"><start_date>{$x}
                </start_date></emp>',
               emp_id,
               start_date AS x
             )
FROM employee
```

Usage

Computed constructors as defined in the XQuery specification are not supported by XMLGEN.

When you execute a query that contains XMLGEN, 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" [*ASA SQL User's Guide*, page 544].

Standards and compatibility

- Part of the SQL/XML draft standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

See also

- "Using the XMLGEN function" [*ASA SQL User's Guide*, page 549]

Example

The following example generates an <emp> element, as well as <last_name>, <first_name>, and <start_date> elements for each employee.

```
SELECT XMLGEN ( '<emp emp_id="{$emp_id}">
                  <last_name>="{$emp_lname}"</last_name>
                  <first_name>="{$emp_fname}"</first_name>
                  <start_date>="{$start_date}"</start_date>
               </emp>',
               emp_id,
               emp_lname,
               emp_fname,
               start_date
             ) AS employee_list
FROM employee
```

YEAR function [Date and time]

Function

Takes a timestamp value as a parameter and returns the year specified by that timestamp as a short value.

Syntax

YEAR (*datetime-expression* **)**

Parameters

datetime-expression A date, time, or timestamp.

Standards and compatibility

- **SQL/92** Vendor extension.

- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following example returns the value 2001.

```
SELECT YEAR ( '2001-09-12')
```

YEARS function [Date and time]

Function

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 131 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 134 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 131 instead for this purpose.

Syntax 1 returns an integer. Syntax 2 returns a timestamp.

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 on casting data types, see "CAST function [Data type conversion]" on page 112.

Usage

The value of YEARS is calculated from the number of first days of the year between the two dates.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

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]

Function

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-expression,
integer-expression,
integer-expression **)**

Parameters

integer-expression The year.

integer-expression The number of the month. If the month is outside the range 1–12, the year is adjusted accordingly.

integer-expression The day number. The day is allowed to be any integer, the date is adjusted accordingly.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Compatible with Adaptive Server Enterprise

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

About this chapter

This chapter presents detailed descriptions of SQL statements in alphabetical order.

Contents

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 53, "Search conditions" on page 23, "Expressions" on page 16, or "Strings" on page 9.

- **column-name** An identifier that represents the name of a column.
- **condition** An expression that evaluates to TRUE, FALSE, or UNKNOWN.
- **connection-name** A string representing the name of an active connection.
- **data-type** A storage data type.
- **expression** An expression.
- **filename** A string containing a filename.
- **hostvar** A C language variable, declared as a host variable preceded by a colon. See "Using host variables" [*ASA Programming Guide*, page 153] for more information.
- **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.
- **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.

- **role-name** An identifier representing the role name of a foreign key.
- **savepoint-name** An identifier that represents the name of a savepoint.
- **search-condition** A condition that evaluates to TRUE, FALSE, or UNKNOWN.
- **special-value** One of the special values described in "Special values" on page 33.
- **statement-label** An identifier that represents the label of a loop or compound statement.
- **table-list** A list of table names, which may include correlation names.

  ☞ For more information, see "FROM clause" on page 491.
- **table-name** An identifier that represents the name of a table.
- **userid** An identifier representing a user name.
- **variable-name** An identifier that represents a variable name.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in upper case, 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.

- **Repeating items** An item that can be repeated is follwed 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/92 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]

Description

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.

Usage

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 433
- "The SQL descriptor area (SQLDA)" [*ASA Programming Guide*, page 181]

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

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 employee';
   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

Description

Use this statement to upgrade a database created with previous versions of the software; or to add Java or jConnect support to any database.

Syntax 1

ALTER DATABASE
[**UPGRADE** [**JAVA** { **ON** | **OFF** | **JDK** { **'1.1.8'** | **'1.3'** } }]
   [**JCONNECT** { **ON** | **OFF** }]
| **REMOVE JAVA**]

Syntax 2

ALTER DATABASE
{   **CALIBRATE** [**SERVER**]
 | **CALIBRATE DBSPACE** *dbspace-name*
 | **CALIBRATE DBSPACE TEMPORARY**
 | **RESTORE DEFAULT CALIBRATION**
}

Syntax 3

ALTER DATABASE *dbfile*
MODIFY [**TRANSACTION**] **LOG**
{ { **OFF** | **ON** } { *log-name* | *log-name* **MIRROR** *mirror-name* | **MIRROR**
        mirror-name } }
[**KEY** *key*]

Parameters

JAVA clause   Controls support for Java in the upgraded database.

- Specify **JAVA ON** to enable support for Java in the database by adding entries for the default Sybase runtime Java classes to the system tables. If Java in the database is already installed, but is at a lower version than the default classes, it is upgraded to the current default classes. The default classes are the JDK 1.3 classes.

- Specify **JAVA OFF** to prevent the addition of Java in the database to databases that do not have already it installed. For databases that already have Java in the database installed, setting **JAVA OFF** does *not* remove Java support: the version of Java remains at the current version. If you wish to remove Java in the database, use the **REMOVE JAVA** clause.

- Specify **JAVA JDK '1.1.8'** or **JAVA JDK '1.3'** to explicitly install support for the named version of the JDK. You can upgrade your database to a higher version of JDK, but you cannot downgrade.

  For JDK 1.1.8 the classes are held *java\1.1\classes.zip* under your SQL Anywhere directory. For JDK 1.3, they are held in *java\1.3\rt.jar.*

The default behavior is **JAVA OFF**.

If you add Java in the database, you must restart the database before it can be used.

☞ For more information, see “Introduction to Java in the Database” [*ASA Programming Guide,* page 51].

Separately licensable option required
Java in the database is a separately licensable component and must be ordered before you can install it.

☞ To order this component, see “Separately-licensable components” [*Introducing SQL Anywhere Studio,* page 5].

JCONNECT clause If you wish to use the Sybase jConnect JDBC driver to access system catalog information, you need to specify JCONNECT ON. If you wish to exclude the jConnect system objects, specify JCONNECT OFF. You can still use JDBC, as long as you do not access system catalog information. The default is to include jConnect support (JCONNECT ON).

Setting JCONNECT OFF does not remove jConnect support from a database.

REMOVE JAVA clause Removes Java in the database from a database. The operation leaves the database as if it were created with JAVA OFF. Java in the database must not be in use when the statement is issued. You must remove all Java classes from the database before executing this statement. The statement does not remove stored procedures and triggers that reference Java objects, and the presence of these objects does not trigger an error in the ALTER DATABASE statement.

Usage

Syntax 1 You can use the ALTER DATABASE statement as an alternative to the Upgrade utility to upgrade a database. After using ALTER DATABASE UPGRADE, you should shut down the database. (The Upgrade utility does this for you automatically.)

Backup before upgrading
As with any software, it is recommended that you make a backup of your database before upgrading.

ALTER DATABASE can be used to upgrade databases created with earlier versions of the software. This applies to maintenance releases as well, so you can upgrade a database created with, for example, version 7.0.2 to 7.0.3 standards using the ALTER DATABASE statement in version 7.0.3 of the software.

In general, changes in databases between minor versions are limited to additional database options and minor system table changes.

When used to upgrade a database, ALTER DATABASE makes the following changes:

- ♦ Upgrades the system tables to the current version.
- ♦ Adds any new database options.
- ♦ Drops and recreates all system stored procedures.

You can also use ALTER DATABASE to just add Java in the database or jConnect features if the database was created with the current version of the software.

☞ For more information on adding Java support, see "Java-enabling a database" [*ASA Programming Guide,* page 87]. For more information on adding jConnect support to a Version 6 database, see "Installing jConnect system objects into a database" [*ASA Programming Guide,* page 111].

Not all features made available

Features that require a physical reorganization of the database file are not made available by ALTER DATABASE. Such features include index enhancements and changes in data storage. To obtain the benefits of these enhancements, you must unload and reload your database.

For more information, see "Rebuilding databases" [*ASA SQL User's Guide,* page 580].

Syntax 2 You can also use ALTER DATABASE 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.

In normal operation, the cost model uses a built-in default DTT model. This default model was designed based on typical hardware and configuration. In rare cases when you are using specialized hardware such as non-standard disk drives, and when you are having performance problems, it may be useful to overwrite the default model with one based on your particular setup. However, it is generally recommended to leave the default in place.

When you recalibrate the I/O cost model, the server is unavailable for other use. In addition, it is essential that all other activities on the computer are idle. Recalibrating the server is an expensive operation and may take some time to complete.

When you use the CALIBRATE [SERVER] argument, all dbspaces are calibrated except for the temporary dbspace. Use CALIBRATE TEMPORARY DBSPACE to calibrate it. Use CALIBRATE DBSPACE *dbspace-name* to calibrate a single dbspace. Use RESTORE DEFAULT CALIBRATION to restore the default DTT model.

Syntax 3 You can use the ALTER DATABASE statement to change the transaction log and mirror 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.

Permissions

Must have DBA authority, and must be the only connection to the database.

For REMOVE JAVA, Java in the database must not be in use when the statement is issued.

ALTER DATABASE UPGRADE is not supported on Windows CE.

Separately licensable option required
Java in the database is a separately licensable component and must be ordered before you can install it.

☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

Side effects

Automatic commit

See also

- ♦ "CREATE DATABASE statement" on page 338
- ♦ "The Upgrade utility" [*ASA Database Administration Guide*, page 599]
- ♦ "CREATE STATISTICS statement" on page 399
- ♦ "The Transaction Log utility" [*ASA Database Administration Guide*, page 580]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following example upgrades a database to enable Java operations.

```
ALTER DATABASE UPGRADE
JAVA ON
```

The following example sets the transaction log filename associated with *asademo.db* to *newdemo.log*.

```
ALTER DATABASE 'asademo.db'
MODIFY LOG ON 'newdemo.log'
```

ALTER DBSPACE statement

Description

Use this statement to pre-allocate space for a dbspace or for the transaction log, or when a database file is renamed or moved.

Syntax

ALTER DBSPACE { *dbspace-name* | **TRANSLOG** | **TEMPORARY** }
{ **ADD** *number* [**PAGES** | **KB** | **MB** | **GB** | **TB**]
| **RENAME** *filename-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 256K at a time for page sizes of 1K, 2K, 4K and 8K, 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.

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 Adaptive Server Anywhere finds the new file when the database is started.

Using ALTER DBSPACE with RENAME on the main dbspace, SYSTEM, has no effect.

Usage

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 SYSFILE 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 Adaptive Server Anywhere where to find the main database file. The main database file holds the system tables. Adaptive Server Anywhere looks in these system tables to find the location of the other dbspaces, and then opens each of the other dbspaces.

Permissions

Must have DBA authority. Must be the only connection to the database.

Side effects

Automatic commit.

See also

- "CREATE DBSPACE statement" on page 344
- "Working with databases" [*ASA SQL User's Guide*, page 31]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

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 megabytes:

```
ALTER DBSPACE system
ADD 400 MB
```

The following example changes the filename associated with the system_2 dbspace:

```
ALTER DBSPACE system_2
RENAME 'e:\db\dbspace2.db'
```

ALTER DOMAIN statement

Description	Use this statement to rename a user-defined domain or data type. You cannot rename Java types with this statement.
Syntax	**ALTER** { **DOMAIN** | **DATATYPE** } *user-type* **RENAME** *new-name*
Usage	When you execute this statement, the name of the user-defined domain or data type is updated in the SYSUSERTYPE system table. You cannot rename Java types with this statement.

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	"CREATE DOMAIN statement" on page 347 "Domains" on page 77 "Using domains" [*ASA SQL User's Guide*, page 92]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following example renames the Address domain to MailingAddress:

```
ALTER DOMAIN Address RENAME MailingAddress
```

ALTER EVENT statement

Description

Use this statement to change the definition of an event or its associated handler for automating predefined actions. Also, to alter the definition of scheduled actions.

Syntax

ALTER EVENT *event-name*
[**DELETE TYPE** | **TYPE** *event-type*]
{ **WHERE** { *trigger-condition* | **NULL** }
| { **ADD** | **MODIFY** | **DELETE** } **SCHEDULE** *schedule-spec*
}
[**ENABLE** | **DISABLE**]
[[**MODIFY**] **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

DELETE TYPE clause Removes an association of the event with an event type. For a description of event types, see "Choosing a system event" [*ASA Database Administration Guide,* page 307].

ADD | MODIFY | DELETE SCHEDULE clause Changes the definition of a schedule. Only one schedule can be altered in any one ALTER EVENT statement.

WHERE clause The WHERE NULL option deletes a condition.

For descriptions of most of the parameters, see "CREATE EVENT statement" on page 351.

Usage

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 312
- "CREATE EVENT statement" on page 351

ALTER FUNCTION statement

Description

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 following the name

Syntax 2

ALTER FUNCTION [*owner*.]*function-name* **SET HIDDEN**

Usage

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 stored procedure debugger will not show the function definition, nor will it be available through procedure profiling.

Permissions

Must be the owner of the function or be DBA.

Side effects

Automatic commit.

See also

- "CREATE FUNCTION statement" on page 362
- "ALTER PROCEDURE statement" on page 278
- "DROP statement" on page 454
- "Hiding the contents of procedures, functions, triggers and views" [*ASA SQL User's Guide*, page 716]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

ALTER INDEX statement

Description

Use this statement to rename an index or foreign key, or to change the clustered nature of the index.

Syntax

ALTER { *index-spec* { *rename-clause* | *cluster-clause* }
| *foreign-key-spec* { *rename-clause* | *cluster-clause* }
| *primary-key-spec cluster-clause* }

index-spec :
INDEX *index-name* **ON** [*owner* **.**]*table-name*

foreign-key-spec :
[**INDEX**] **FOREIGN KEY** *role-name* **ON** [*owner* **.**]*table-name*

primary-key-spec :
[**INDEX**] **PRIMARY KEY ON** [*owner* **.**]*table-name*

rename-clause :
RENAME [**AS** | **TO**] *index-name*

cluster-clause :
CLUSTERED | **NONCLUSTERED**

Parameters

rename-clause Specify the new name for the index.

cluster-clause Specify whether the index should be changed to CLUSTERED or NONCLUSTERED. Only one index on a table can be clustered.

Usage

The ALTER INDEX statement carries out two tasks:

- It can be used to rename an index or foreign key.
- It can also 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 can be clustered. For more information on clustered indexes, see "CREATE INDEX statement" on page 368.

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 have DBA authority.

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 368

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement renames the index ix_prod_name on the product table to ixProductName:

```
ALTER INDEX ix_prod_name ON product
RENAME TO ixProductName
```

The following statement changes ix_prod_name to be a clustered index:

```
ALTER INDEX ix_prod_name ON product
CLUSTERED
```

ALTER PROCEDURE statement

Description

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.

Syntax 1

ALTER PROCEDURE [*owner*.]*procedure-name*
procedure-definition

procedure-definition:
CREATE PROCEDURE syntax following the name

Syntax 2

ALTER PROCEDURE [*owner*.]*procedure-name*
REPLICATE { **ON** | **OFF** }

Syntax 3

ALTER PROCEDURE [*owner*.]*procedure-name* **SET HIDDEN**

Usage

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 2 of the ALTER PROCEDURE statement has the same effect as the sp_setreplicate or sp_setrepproc 'table' Adaptive Server Enterprise system procedures.

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 stored procedure 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 be DBA.

Side effects

Automatic commit.

See also

- "CREATE PROCEDURE statement" on page 373
- "ALTER FUNCTION statement" on page 275
- "DROP statement" on page 454
- "Hiding the contents of procedures, functions, triggers and views" [*ASA SQL User's Guide*, page 716]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

ALTER PUBLICATION statement

Description

Use this statement to alter a publication. In MobiLink, a publication identifies synchronized data in a Adaptive Server Anywhere remote database. In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax

ALTER PUBLICATION [*owner*.]*publication-name alterpub-clause*, . . .

alterpub-clause:
  ADD TABLE *article-description*
| **MODIFY TABLE** *article-description*
| { **DELETE** | **DROP** } **TABLE** [*owner*.]*table-name*
| **RENAME** *publication-name*

owner , *publication-name*, *table-name* : *identifier*

article-description :
table-name [**(** *column-name*, . . . **)**]
[**WHERE** *search-condition*]
[**SUBSCRIBE BY** *expression*]

Usage

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 specification of the modified article must be entered.

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" on page 385
- "DROP PUBLICATION statement" on page 459
- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 290
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 402
- "sp_add_article procedure" [*SQL Remote User's Guide*, page 381]
- "sp_add_article_col procedure" [*SQL Remote User's Guide*, page 383]

Standards and compatibility

- **SQL/92** Vendor extension.

♦ **SQL/99** Vendor extension.

Example

The following statement adds the customer table to the pub_contact publication.

```
ALTER PUBLICATION pub_contact
   ADD TABLE customer
```

ALTER REMOTE MESSAGE TYPE statement [SQL Remote]

Description

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.

Usage

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 e-mail system, the address string must be a valid e-mail 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.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE REMOTE MESSAGE TYPE statement [SQL Remote]" on page 388
- "sp_remote_type procedure" [*SQL Remote User's Guide*, page 427]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Example

The following statement changes the publisher's address for the FILE message link to **new_addr**.

```
CREATE REMOTE MESSAGE TYPE file
ADDRESS 'new_addr'
```

ALTER SERVER statement

Description

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 :
ASAJDBC | **ASEJDBC**
| **ASAODBC** | **ASEODBC**
| **DB2ODBC** | **MSSODBC**
| **ORAODBC** | **ODBC**

connection-info :
machine-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" [*ASA SQL User's Guide*, page 635].

USING clause The USING clause is specified to change the server connection information. For information about *connection-info*, see "CREATE SERVER statement" on page 392.

CAPABILITY clause The CAPABILITY clause turns a server capability ON or OFF. Server capabilities are stored in the system table syscapability. The names of these capabilities are stored in the system table syscapabilityname. The syscapability table contains no entries for a remote server until the first connection is made to that server. At the first connection, Adaptive Server Anywhere interrogates the server about its capabilities and then populates the syscapability table. For subsequent connections, the 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.

Usage

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

- "CREATE SERVER statement" on page 392
- "DROP SERVER statement" on page 461
- "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635]
- "Troubleshooting remote data access" [*ASA SQL User's Guide*, page 632]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

The following example changes the server class of the Adaptive Server named ase_prod so its connection to Adaptive Server 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

Description

Use this statement to alter a web service.

Syntax 1

ALTER SERVICE *service-name*
[**TYPE 'DISH'**]
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2

CREATE SERVICE *service-name*
[**TYPE 'SOAP'**]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
[**AS** *statement*]

Syntax 3

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 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 may 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 unsecure 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 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.

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 ASADataset 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 are 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.

Usage

The alter service statement causes the database server to act as a web server.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "CREATE SERVICE statement" on page 395, "DROP SERVICE statement" on page 462

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.

- **Sybase** Not supported by Adaptive Server Enterprise.

Example

To quickly set up a web server, start a database server with the -xs switch, then execute the following statements:

```
CREATE SERVICE tables TYPE 'XML'

ALTER SERVICE tables
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTABLE
```

After executing these statements, use any web browser to open the URL http://localhost/tables.

ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Description

Use this statement in an Adaptive Server 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** *protocol*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option*=*value*, . . .]
[**MODIFY OPTION** *option*=*value*, . . .]
[**DELETE** { **ALL OPTION** | **OPTION** *option*, . . . }]

ml_username: *identifier*

protocol-type: **http** | **https** | **https_fips** |**tcpip** | **ActiveSync**

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 for extended options and connection parameters" [*MobiLink Clients*, page 73].

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 Clients*, page 111].

ADDRESS clause This clause specifies network protocol options, including the location of the MobiLink synchronization server.

☞ For a complete list of protocol options, see "CommunicationAddress (adr) extended option" [*MobiLink Clients*, page 106].

ADD OPTION, MODIFY 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.

The values for each option cannot contain the characters "**=**" or "**,**" or "**;**".

☞ For a complete list of options, see "dbmlsync extended options" [*MobiLink Clients*, page 105].

Usage

Use this statement to alter a synchronization subscription within a MobiLink remote or reference database.

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" on page 385
- "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 404

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Examples

Create a default subscription, which contains default subscription values, for the sales publication (by omitting the FOR clause). Indicate the address of the MobiLink synchronization server and specify that only the Certicom root certificate is to be trusted.

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   ADDRESS 'host=test.internal;port=2439;
      security=ecc_tls'
   OPTION memory='2m';
```

Subscribe MobiLink user ml_user1 to the sales publication. Set the memory option to 3 Mb, rather than the value specified in the default publication.

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR 'ml_user1'
   OPTION memory='3m';
```

ALTER SYNCHRONIZATION USER statement [MobiLink]

Description

Use this statement in an Adaptive Server Anywhere remote database to alter the properties of a MobiLink user.

Syntax

ALTER SYNCHRONIZATION USER *ml_username*
[**TYPE** *protocol-type*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option*=*value*, . . .]
[**MODIFY OPTION** *option*=*value*, . . .]
[**DELETE** { **ALL OPTION** | **OPTION** *option* }]

ml_username: *identifier*

protocol-type: **http** | **https** | **https_fips** | **tcpip** | **ActiveSync**

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 Clients,* page 111].

ADDRESS clause This clause specifies network protocol options, including the location of the MobiLink synchronization server.

☞ For a complete list of protocol options, see "CommunicationAddress (adr) extended option" [*MobiLink Clients,* page 106].

ADD OPTION, MODIFY 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 "dbmlsync extended options" [*MobiLink Clients,* page 105].

Usage

Use this statement to alter the properties of a synchronization user within a MobiLink remote database.

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 290
- ♦ “CREATE SYNCHRONIZATION USER statement [MobiLink]” on page 404
- ♦ “CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]” on page 402

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.

ALTER TABLE statement

Description

Use this statement to modify a table definition or to enable a table to take part in Replication Server replication.

Syntax 1

ALTER TABLE [*owner*.]*table-name*
{ *add-clause* | *modify-clause* | *drop-clause* | *rename-clause* }

add-clause :
  ADD { *column-definition* | *table-constraint* }
| { **ADD PCTFREE** *integer* | **PCTFREE DEFAULT** }

modify-clause :
  MODIFY *column-definition*
| **MODIFY** *column-name* { **DEFAULT** *default-value*
  | [**NOT**] **NULL**
  | [**CONSTRAINT** *constraint-name*]
      CHECK { **NULL** | **(** *new-condition* **)** } }
| **ALTER** *column-name* *column-modification*
| **ALTER** *constraint-name* **CHECK (** *new-condition* **)**

drop-clause :
{ **DELETE** | **DROP** }{
   column-name
  | **CONSTRAINT** *constraint-name*
  | **CHECK**
  | **UNIQUE (** *column-name*, . . . **)**
  | **PRIMARY KEY**
  | **FOREIGN KEY** *role-name* }

rename-clause :
  RENAME *new-table-name*
| **RENAME** *column-name* **TO** *new-column-name*
| **RENAME** *constraint-name* **TO** *new-constraint-name*

column-definition :
column-name data-type [[**NOT**] **NULL**] [**DEFAULT** *default-value*] [*column-constraint* . . .]

table-constraint :
[**CONSTRAINT** *constraint-name*] { **UNIQUE (** *column-name*, . . . **)**
  | **PRIMARY KEY** [**CLUSTERED**] **(** *column-name*, . . . **)**
  | *foreign-key-constraint*
  | **CHECK (** *condition* **)** }

column-constraint :
[**CONSTRAINT** *constraint-name*] { **UNIQUE**
  | **PRIMARY KEY**
  | **REFERENCES** *table-name*
      [**(** *column-name* **)**] [*actions*] [**CLUSTERED**]
  | **CHECK (** *condition* **)** }
| **COMPUTE (** *expression* **)**

column-modification :
  SET DEFAULT *default-value*
| **DROP DEFAULT**
| **ADD** [**CONSTRAINT** *column-constraint-name*] **CHECK (** *condition* **)**
| { **DELETE** | **DROP** } **CONSTRAINT** *column-constraint-name*
| { **DELETE** | **DROP** } **CHECK**
| **SET COMPUTE (** *expression* **)**
| **DROP COMPUTE**

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-constraint :
[**NOT NULL**] **FOREIGN KEY** [*role-name*] [**(***column-name*, . . . **)**]
REFERENCES *table-name* [**(***column-name*, . . . **)**] [*actions*]
[**CHECK ON COMMIT**] [**CLUSTERED**]

actions :
[**ON UPDATE** *action*] [**ON DELETE** *action*]

action :
CASCADE | **SET NULL** | **SET DEFAULT** | **RESTRICT**

Syntax 2

ALTER TABLE [*owner*.]*table-name* **REPLICATE** { **ON** | **OFF** }

Parameters

add-clause Add a new column or table constraint to the table. For more information, see below.

modify-clause Change a single column definition. For more information, see below.

drop-clause Drop a column or a table constraint. For more information, see below.

rename-clause Change the name of the table, a column, or a constraint. For more information, see below.

ADD column-definition Add a new column to the table.

If the column has a default value, all rows of the new column are populated with that default value.

Adaptive Server Anywhere optimizes the creation of columns that are allowed to contain NULL. The first column allowed to contain NULL allocates room for eight such columns, and initializes all eight to be NULL. (This requires no extra storage.) Thus, the next seven columns added require no changes to the rows of the table. Adding a ninth column then allocates room for another eight such columns and modifies each row of the table to allocate the extra space.

ADD table-constraint Add a constraint to the table. See "CREATE TABLE statement" on page 407 for a full explanation of table constraints.

If PRIMARY KEY is specified, the table must not already have a primary key that was created by the CREATE TABLE statement or another ALTER TABLE statement.

The optional constraint name allows you to modify or drop individual constraints at a later time, rather than having to modify the entire table constraint.

Table CHECK constraints fail only if a value of FALSE is returned. A value of UNKNOWN allows the change.

PCTFREE Specify 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.

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 if DEFAULT is specified, 200 bytes are reserved in each page.

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 or the table is dropped.

The PCTFREE specification can be used for base, global temporary, or local temporary tables. Except for local temporary tables, the value for PCTFREE is stored in the SYSATTRIBUTE system table.

☞ For more information, see "SYSATTRIBUTE system table" on page 678.

MODIFY column-definition Change the length or data type of an existing column in a table. If NOT NULL is specified, a NOT NULL constraint is added to the named column. Otherwise, the NOT NULL constraint for the column will not be changed. If necessary, the data in the modified column will be converted to the new data type. If a conversion error occurs, the operation will fail and the table will be left unchanged.

You cannot modify a column to make it a computed column using the MODIFY clause. Computed columns can only be added, dropped, or altered using the ADD, DELETE/DROP, or ALTER clause.

Deleting an index, constraint, or key

If the column is contained in a uniqueness constraint, a foreign key, or a primary key, then the constraint or key must be deleted before the column can be modified. If a primary key is deleted, all foreign keys referencing the table will also be deleted.

You cannot MODIFY a table or column constraint. To change a constraint, you must DELETE the old constraint and ADD the new constraint.

MODIFY column-name DEFAULT default-value Change the default value of an existing column in a table. To remove a default value for a column, specify DEFAULT NULL. Modifying a default value does not change any existing values in the table.

ALTER column-name column-modification Change the definition of a column. The permitted modifications are as follows:

- **SET DEFAULT default-value** Change the default value of an existing column in a table. You can also use the MODIFY clause for this task, but ALTER is SQL/92 compliant, and MODIFY is not. Modifying a default value does not change any existing values in the table.

- **DROP DEFAULT** Remove the default value of an existing column in a table. You can also use the MODIFY clause for this task, but ALTER is SQL/92 compliant, and MODIFY is not. Dropping a default does not change any existing values in the table.

- **ADD** Add a named constraint or a CHECK condition to the column. The new constraint or condition applies only to operations on the table after its definition. The existing values in the table are not validated to confirm that they satisfy the new constraint or condition.

- **CONSTRAINT column-constraint-name** The optional column constraint name allows you to modify or drop individual constraints at a later time, rather than having to modify the entire column constraint.

- **SET COMPUTE (expression)** Change the expression associated with a computed column. The values in the column are recalculated when the statement is executed, and the statement fails if the new expression is invalid.

- **DROP COMPUTE** Change a column from being a computed column to being a non-computed column. This statement does not change any existing values in the table.

MODIFY column-name [NOT] NULL Change the NOT NULL constraint on the column to allow or disallow NULL values in the column.

MODIFY column-name CHECK NULL Delete the check constraint for the column. This statement cannot be used on databases created before version 5.0.

MODIFY column-name CHECK (condition) Replace the existing CHECK condition for the column with the one specified. This statement cannot be used on databases created before version 5.0.

DELETE column-name Delete the column from the table. If the column is contained in any index, uniqueness 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.

DELETE constraint-name Delete the named constraint from the table definition.

DELETE CHECK Delete all check constraints for the table. This includes both table check constraints and column check constraints.

DELETE UNIQUE (column-name, . . .) Delete a uniqueness constraint for this table. Any foreign keys referencing this uniqueness constraint (rather than the primary key) will also be deleted.

DELETE PRIMARY KEY Delete the primary key constraint for this table. All foreign keys referencing the primary key for this table will also be deleted.

DELETE FOREIGN KEY role-name Delete the foreign key constraint for this table with the given role name.

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. Foreign keys that were automatically assigned the old table name will not change names.

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.

Usage

Syntax 1 The ALTER TABLE statement changes table attributes (column definitions, constraints) in a table that was previously created. Note that the syntax allows a list of alter clauses; however, only one table-constraint or column-constraint can be added, modified or deleted in one ALTER TABLE statement. A table cannot be both added and modified in the same statement.

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 server will not process requests referencing the table while the statement is being processed.

☞ For more information on using the CLUSTERED option, see "Using clustered indexes" [*ASA SQL User's Guide*, page 63].

Before version 5.0, all table and column constraints were held in a single table constraint. Consequently, for these databases individual constraints on columns cannot be deleted using the MODIFY column-name CHECK NULL clause or replaced using the MODIFY column-name CHECK (condition) clause. To use these statements, the entire table constraint should be deleted and the constraints added back using the MODIFY column-name CHECK (condition) clause. At this point you can use MODIFY CHECK.

Syntax 2 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. The remainder of this section describes syntax 1.

Permissions

Must be one of the following:

- ♦ The owner of the table.
- ♦ A user with DBA authority.
- ♦ A user 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.

Side effects

Automatic commit.

The MODIFY and DELETE (DROP) options close all cursors for the current connection.

A checkpoint is carried out at the beginning of the ALTER TABLE operation.

Once you alter a column or table, any stored procedures, views or other items that refer to the altered column no longer work.

If you change the declared length of a column, the statistics for that column are dropped.

See also

- "CREATE TABLE statement" on page 407
- "DROP statement" on page 454
- "SQL Data Types" on page 53
- "Altering tables" [*ASA SQL User's Guide,* page 41]
- "Special values" on page 33
- "Using table and column constraints" [*ASA SQL User's Guide,* page 88]
- "ALLOW_NULLS_BY_DEFAULT option [compatibility]" [*ASA Database Administration Guide,* page 633]

Standards and compatibility

- **SQL/92** Intermediate-level feature. MODIFY is not SQL/92 compliant.
- **SQL/99** ADD COLUMN is a core feature. Other clauses are vendor extensions or implementation of specific, named extensions to SQL/99.
- **Sybase** Some clauses are supported by Adaptive Server Enterprise.

Example

The following example adds a new column to the employee table showing which office they work in.

```
ALTER TABLE employee
ADD office CHAR(20) DEFAULT 'Boston'
```

The following example drops the office column from the employee table.

```
ALTER TABLE employee
DELETE office
```

The address column in the customer table can currently hold up to 35 characters. To allow it to hold up to 50 character, type the following.

```
ALTER TABLE customer
MODIFY address CHAR(50)
```

The following example adds a column to the customer table assigning each customer a sales contact.

```
ALTER TABLE customer
ADD sales_contact INTEGER
REFERENCES employee (emp_id)
ON UPDATE CASCADE
ON DELETE SET NULL
```

This foreign key is constructed with cascading updates and is set 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

Description	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 following the trigger name
Syntax 2	**ALTER TRIGGER** *trigger-name* **ON** [*owner.*] *table-name* **SET HIDDEN**

Usage

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 419 and "CREATE TRIGGER statement [T-SQL]" on page 426.

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.

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 stored procedure debugger will not show the trigger definition, nor will it be available through procedure profiling.

Permissions

Must be the owner of the table on which the trigger is defined, or be DBA, or have ALTER permissions on the table and have RESOURCE authority.

Side effects

Automatic commit.

See also

- ♦ "CREATE TRIGGER statement" on page 419
- ♦ "CREATE TRIGGER statement [T-SQL]" on page 426
- ♦ "DROP statement" on page 454
- ♦ "Hiding the contents of procedures, functions, triggers and views" [*ASA SQL User's Guide*, page 716]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

ALTER VIEW statement

Description

Use this statement to replace a view definition with a modified version. If SET HIDDEN or RECOMPILE is used, the view definition is not specified.

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** }

Usage

Syntax 1 The ALTER VIEW statement is identical in syntax to the CREATE VIEW statement except for the first word. The ALTER VIEW statement replaces the entire contents of the CREATE VIEW statement with the contents of the ALTER VIEW statement. Existing permissions on the view are maintained, and do not have to be reassigned. If a DROP VIEW followed by a CREATE VIEW is used, instead of ALTER VIEW, permissions on the view would have to be reassigned.

Syntax 2 You can use SET HIDDEN to scramble the definition of the associated view and cause it to become unreadable. The view 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 stored procedure debugger will not show the view definition, nor will it be available through procedure profiling.

The RECOMPILE clause is used to re-create the column definitions for an existing view. If the tables a view is based on are updated, you can use RECOMPILE to update the view definition. RECOMPILE is also used by the sa_recompile_views system procedure to update view definitions when rebuilding a database.

☞ For more information, see "sa_recompile_views system procedure" on page 825.

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.

☞ For information on the keywords and options, see "CREATE VIEW statement" on page 428.

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 428
- ♦ "DROP statement" on page 454
- ♦ "Hiding the contents of procedures, functions, triggers and views" [*ASA SQL User's Guide*, page 716]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

ALTER WRITEFILE statement (deprecated)

Description

Use this statement to change the name of the read-only database file to which a write file refers.

Deprecated statement
The use of write files is deprecated.

Syntax

ALTER WRITEFILE *write-file-name*
REFERENCES *db-file-name* [**KEY** *key*]

write-file-name | *db-file-name* : *string*

Usage

The ALTER WRITEFILE statement changes the name of the read-only database file to which the write file refers. If you move the database file from one directory to another, you can use this statement to point the write file to the new location.

The path name of the database file is relative to the database server's current directory at startup.

☞ For information on escaping backslash characters in strings, see "Strings" on page 9.

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.

You need to specify a KEY value if you want to change the writefile for a strongly encrypted database.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "CREATE WRITEFILE statement (deprecated)" on page 430
- "The Write File utility (deprecated)" [*ASA Database Administration Guide*, page 609]
- "Working with write files (deprecated)" [*ASA Database Administration Guide*, page 294]
- "Using the utility database" [*ASA Database Administration Guide*, page 296]
- "DatabaseKey connection parameter [DBKEY]" [*ASA Database Administration Guide*, page 187]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement changes the existing write file *c:\readwrite.wrt* to point to the database file *h:\readonly.db*.

```
ALTER WRITEFILE ’c:\\readwrite.wrt’
REFERENCES ’h:\\readonly.db’
```

BACKUP statement

Description

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

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 those files, relative to the server's current directory at startup. If the directory does not already exist, it is created. Specifying an empty string as a directory allows you to rename or truncate the log without making a copy of it first.

WAIT BEFORE START clause This clause ensures 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 This clause may be used 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 This clause may be used to cause backup copies of the main database file and any associated dbspaces to be made. The transaction log is not copied.

TRANSACTION LOG ONLY clause This clause may be used 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 server to rename the current transaction log at the completion of the backup. The name for the renamed copy will be of the form YYMMDDnn.log. 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 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 Windows NT or NetWare, the first tape drive is *\\.\tape0*.

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 9.

ON EXISTING ERROR 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 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 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.

Each BACKUP operation, whether image or archive, updates a history file called *backup.syb.*

☞ For more information about the *backup.syb* file, see "ASLOGDIR environment variable" [*ASA Database Administration Guide*, page 278].

WITH COMMENT Record a comment in the archive file and in the backup history file.

HISTORY By default, each BACKUP 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 backups occur frequently
- there is no procedure to periodically archive or delete the *backup.syb* file
- disk space is very limited

Usage

The first syntax is an image backup and the second syntax is an archive backup.

Syntax 1 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 machine, but you can specify the -s option to create the backup on the server when using the Backup utility. In the case of the BACKUP statement, however, the backup can only be made on the 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 the chapter "Backup and Data Recovery" [*ASA Database Administration Guide*, page 373].

Syntax 2 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. Archive backups to tape are not supported on versions of NetWare earlier than NetWare 5.

There can be only one backup on a given tape. The file *backup.syb* records the BACKUP and RESTORE operations that have been performed on a given server.

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 statement references an archive file containing only a transaction log, the statement must still specify a filename 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 statement looks like:

```
RESTORE DATABASE 'c:\mynewdb\my.db' FROM
archive-location
```

Permissions

Must have DBA authority.

Side effects

Causes a checkpoint.

See also

- "The Backup utility" [*ASA Database Administration Guide*, page 498]
- "RESTORE DATABASE statement" on page 580
- "Backup and Data Recovery" [*ASA Database Administration Guide*, page 373]
- "EXECUTE IMMEDIATE statement [SP]" on page 475

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not compatible with Adaptive Server Enterprise.
- **Windows CE** Only the BACKUP DATABASE DIRECTORY syntax (syntax 1 above) is supported on the Windows CE platform.

Example

Back up the current database and the transaction log to a file, renaming 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 statement with a dynamically-constructed filename:

```
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

Description

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 "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655].

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.

Usage

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 436
- ♦ "DECLARE LOCAL TEMPORARY TABLE statement" on page 443
- ♦ "LEAVE statement" on page 535
- ♦ "SIGNAL statement" on page 622
- ♦ "RESIGNAL statement" on page 579
- ♦ "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]
- ♦ "Atomic compound statements" [*ASA SQL User's Guide*, page 680]

Standards and compatibility

- ♦ **SQL/92** Persistent Stored Module feature.
- ♦ **SQL/99** Persistent Stored Module feature.
- ♦ **Sybase** Supported by Adaptive Server Enterprise. This does not mean that all statements inside a compound statement are supported.

  The BEGIN and END keywords are not required in Transact-SQL.

  BEGIN and END are used in Transact-SQL to group a set of statements into a single compound statement, so that control statements such as IF . . . ELSE, which only affect the execution of a single SQL statement, can affect the execution of the whole group. The ATOMIC keyword is not supported by Adaptive Server Enterprise.

  In Transact-SQL. DECLARE statements need not immediately follow a BEGIN keyword, and the cursor or variable that is declared exists for the duration of the compound statement. You should declare variables at the beginning of the compound statement for compatibility.

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 company_name, CAST(
            sum(sales_order_items.quantity *
            product.unit_price) AS INTEGER) VALUE
      FROM customer
            LEFT OUTER JOIN sales_order
            LEFT OUTER JOIN sales_order_items
            LEFT OUTER JOIN product
      GROUP BY company_name;
   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

Description

Use this statement to begin a user-defined transaction.

Syntax

BEGIN TRAN[**SACTION**] [*transaction-name*]

Usage

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 Adaptive Server 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/92 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 Adaptive Server Anywhere is ON, in which case Adaptive Server Anywhere runs in chained mode. (ODBC users should also check the AutoCommit ODBC setting). The default for TDS connections is OFF.

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.

☞ For more information, see “CHAINED option [compatibility]” [*ASA Database Administration Guide*, page 640].

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 329
- ♦ "ISOLATION_LEVEL option [compatibility]" [*ASA Database Administration Guide*, page 659]
- ♦ "ROLLBACK statement" on page 591
- ♦ "SAVEPOINT statement" on page 596
- ♦ "Savepoints within transactions" [*ASA SQL User's Guide*, page 104]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Supported by Adaptive Server Enterprise.

Example

The following batch reports successive values of @@trancount as 0, 1, 2, 1, and 0. The values are printed on the server 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. Adaptive Server Anywhere does not set the @@trancount value to 1 when a transaction is started implicitly. Consequently, the Adaptive Server 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 Adaptive Server 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 Adaptive Server Anywhere, and Adaptive Server Enterprise after the BEGIN TRANSACTION statement.

CALL statement

Description	Use this statement to invoke a procedure.
Syntax 1	[*variable* =] **CALL** *procedure-name* **(** [*expression*, . . .] **)**
Syntax 2	[*variable* =] **CALL** *procedure-name* **(** [*parameter-name* = *expression*, . . .] **)**

Usage

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 values will be 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 or. 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" [*ASA SQL User's Guide*, page 686]).

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, ... )
```

☞ For information on returning non-integer values, see "CREATE FUNCTION statement" on page 362.

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

- "CREATE PROCEDURE statement" on page 373
- "GRANT statement" on page 503
- "EXECUTE statement [T-SQL]" on page 473
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.

- **Sybase** Not supported by Adaptive Server Enterprise. For an alternative that is supported, see "EXECUTE statement [T-SQL]" on page 473.

Example

Call the sp_customer_list procedure. This procedure has no parameters, and returns a result set.

```
  CALL sp_customer_list()
```

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("DBA".sales_order.id)
   INTO Orders
   FROM "DBA".customer
   KEY LEFT OUTER JOIN "DBA".sales_order
   WHERE "DBA".customer.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

Description

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

Usage

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.

For more information about the ISNULL function, see "ISNULL function [Data type conversion]" on page 170.

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
```

☞ For more information about NULL values, see "Unknown Values: NULL" [*ASA SQL User's Guide*, page 232].

CASE statement is different from CASE expression
Do not confuse the syntax of the CASE statement with that of the CASE expression.

For information on the CASE expression, see "CASE expressions" on page 18.

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 312
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- **SQL/92** Persistent Stored Module feature. Adaptive Server Anywhere supports the CASE statement allowing WHEN NULL. This is a vendor extension to the SQL/92 standard.
- **SQL/99** Persistent Stored Module feature. Adaptive Server Anywhere supports the CASE statement allowing WHEN NULL. This is a vendor extension to the SQL/92 standard.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following procedure using a case statement classifies the products listed in the product table of the 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 "DBA"."product"
   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 sample database.

```
CREATE PROCEDURE StockLevel (IN product_id INT)
BEGIN
   DECLARE qty INT;
   SELECT quantity INTO qty FROM product
   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

Description	Use this statement to checkpoint the database.
Syntax	**CHECKPOINT**

Usage

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 issue the CHECKPOINT statement.

☞ For a full description of checkpoints, see "Backup and Data Recovery" [*ASA Database Administration Guide*, page 373].

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

- "CHECKPOINT_TIME option [database]" [*ASA Database Administration Guide*, page 641]
- "RECOVERY_TIME option [database]" [*ASA Database Administration Guide*, page 686]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Adaptive Server Enterprise.

CLEAR statement [Interactive SQL]

Description	Use this statement to clear the Interactive SQL panes.
Syntax	**CLEAR**
Usage	The CLEAR statement is used to clear the SQL Statements pane, the Messages pane and the Results, Messages, Plan, and UltraLite Plan tabs in the Results pane.
Permissions	None.
Side effects	Closes the cursor associated with the data being cleared.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not applicable

CLOSE statement [ESQL] [SP]

Description

Use this statement to close a cursor.

Syntax

CLOSE *cursor-name*

cursor-name : *identifier* | *hostvar*

Usage

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 553
- "DECLARE CURSOR statement [ESQL] [SP]" on page 436
- "PREPARE statement [ESQL]" on page 563

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Adaptive Server Enterprise.

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 company_name, CAST(    sum(sales_order_items.quantity
         *
   product.unit_price) AS INTEGER) VALUE
   FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
   GROUP BY company_name;
   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

Description

Use this statement to store a comment in the system tables for a database object.

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*
  | **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*
}
IS *comment*

comment : *string* | **NULL**

Usage

Several system tables have a column named Remarks that allows you to associate a comment with a database item (SYSUSERPERM, SYSTABLE, SYSCOLUMN, SYSINDEX, SYSLOGIN, SYSFOREIGNKEY, SYSPROCEDURE, SYSTRIGGER). The COMMENT ON statement allows you to set the Remarks column in these system tables. A comment can be removed 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.

Permissions

Must either be the owner of the database object being commented, or have DBA authority.

Side effects

Automatic commit.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following examples show how to add and remove a comment.

Add a comment to the employee table.

```
COMMENT
ON TABLE employee
IS ’Employee information’
```

Remove the comment from the employee table.

```
COMMENT
ON TABLE employee
IS NULL
```

COMMIT statement

Description

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 Adaptive Server Anywhere, see "BEGIN TRANSACTION statement" on page 315. For more information on savepoints, see "SAVEPOINT statement" on page 596.

☞ You can use a set of options to control the detailed behavior of the COMMIT statement. For information, see "COOPERATIVE_COMMIT_TIMEOUT option [database]" [*ASA Database Administration Guide*, page 645], "COOPERATIVE_COMMITS option [database]" [*ASA Database Administration Guide*, page 645], "DELAYED_COMMITS option [database]" [*ASA Database Administration Guide*, page 651], and "DELAYED_COMMIT_TIMEOUT option [database]" [*ASA Database Administration Guide*, page 650]. You can use the Commit connection property to return the number of Commits on the current connection.

Usage

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 clause, 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.

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

- "BEGIN TRANSACTION statement" on page 315
- "PREPARE TO COMMIT statement" on page 565
- "ROLLBACK statement" on page 591

Standards and compatibility

- **SQL/92** Syntax 1 is an entry-level feature. Syntax 2 is a Transact-SQL extension.
- **SQL/99** Syntax 1 is a core feature. Syntax 2 is a Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

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]

Description	Use this statement to open the Interactive SQL Options dialog.
Syntax	**CONFIGURE**

Usage

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. If you select Make Permanent, the options are saved and will be used in subsequent Interactive SQL sessions. If you do not choose Make Permanent, and instead click OK, the options are set temporarily and remain in effect for the current database connection only.

Permissions

None.

Side effects

None.

See also

- “SET OPTION statement” on page 613

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

CONNECT statement [ESQL] [Interactive SQL]

Description

Use this statement to establish a connection to a database.

Syntax 1

CONNECT
[**TO** *engine-name*]
[**DATABASE** *database-name*]
[**AS** *connection-name*]
[**USER**] *userid* **IDENTIFIED BY** *password*

engine-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" [*ASA Database Administration Guide*, page 176].

Usage

The CONNECT statement establishes a connection to the database identified by *database-name* running on the server identified by *engine-name*.

Embedded SQL behavior In Embedded SQL, if no *engine-name* is specified, the default local database server will be assumed (the first database server started). If no *database-name* is specified, the first database on the given server will be 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.

You can connect without explicitly specifying a password by using a host variable for the password and setting the value of the host variable to be the null pointer.

If you are connected to a user ID with DBA authority, you can connect to another user ID without specifying a password. (The output of dbtran requires this capability.)

☞ For a detailed description of the connection algorithm, see "Troubleshooting connections" [*ASA Database Administration Guide*, page 66].

Note
For Adaptive Server 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.

In the user interface, if the password or the user ID and password are not specified, the user is prompted to type the missing information.

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.

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.

Permissions

None.

Side effects

None.

See also

- ♦ "GRANT statement" on page 503
- ♦ "DISCONNECT statement [ESQL] [Interactive SQL]" on page 453
- ♦ "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 610
- ♦ "SETUSER statement" on page 620
- ♦ "Connection parameters" [*ASA Database Administration Guide*, page 176]

Standards and compatibility

- ♦ **SQL/92** Syntax 1 is a full SQL feature. Syntax 2 is a vendor extension.
- ♦ **SQL/99** Syntax 1 is a SQL/foundation feature outside of core SQL. Syntax 2 is a vendor extension.
- ♦ **Sybase** Open Client Embedded SQL supports a different syntax for the CONNECT statement.

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 sample database (*asademo.db*) 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 the DBA from Interactive SQL.

```
CONNECT
TO asademo
USER DBA
IDENTIFIED BY SQL
```

Connect to the sample database using a connect string, from Interactive SQL.

```
CONNECT
USING ’UID=DBA;PWD=SQL;DBN=asademo’
```

Once you connect to the sample database, the database name, your user ID, and the server name appear on the title bar: asademo (DBA) on asademo9.

CREATE COMPRESSED DATABASE statement (deprecated)

Description

Use this statement to create a compressed database from an existing database file, or to expand a compressed database.

Deprecated statement
The use of compressed databases is deprecated.

Syntax

CREATE [**COMPRESSED** | **EXPANDED**] **DATABASE** *new-db-file-name*
FROM *old-db-file-name* [**KEY** *key*]

Usage

Creates a compressed database file from an uncompressed database file, or an uncompressed database file from a compressed one.

Any relative path is resolved relative to the current working directory of the server.

You cannot use this statement on files other than the main database file.

Caution
Compressing an encrypted database removes encryption from the database.

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 operating system account under which the server is running must have write permissions on the directories where files are created.
- The old database file must not be currently running.
- Not supported on Windows CE.
- You must specify a key if you want to create a compressed database for a strongly encrypted database.

Side effects

An operating system file is created.

See also

- "The Compression utility (deprecated)" [*ASA Database Administration Guide*, page 507]
- "The Uncompression utility (deprecated)" [*ASA Database Administration Guide*, page 585]
- "DatabaseKey connection parameter [DBKEY]" [*ASA Database Administration Guide*, page 187]

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement creates a compressed database file named *compress.db* in the *C:\* directory from a database file named *full.db* in the current working directory of the server.

```
CREATE COMPRESSED DATABASE 'C:\\compress.db'
FROM 'full.db'
```

The following statement creates an uncompressed database file named *full.db* in the *C:\* directory from a compressed database file named *compress.db* in the current working directory of the server.

```
CREATE EXPANDED DATABASE 'C:\\full.db'
FROM 'compress.db'
```

CREATE DATABASE statement

Description

Use this statement to create a database. The database is stored as an operating system file.

Syntax

CREATE DATABASE *db-file-name*
 [[**TRANSACTION**] { **LOG OFF** | **LOG ON** } [*log-file-name-string*]
  [**MIRROR** *mirror-file-name-string*]]
[**CASE** { **RESPECT** | **IGNORE** }]
[**PAGE SIZE** *page-size*]
[**COLLATION** *collation-label*]
[**ENCRYPTED** { **ON** | **OFF** | *key-spec* }]
[**BLANK PADDING** { **ON** | **OFF** }]
[**ASE** [**COMPATIBLE**]]
[**JAVA** { **ON** | **OFF** | **JDK** { **'1.1.8'** | **'1.3'** } }]
[**JCONNECT** { **ON** | **OFF** }]
[**PASSWORD CASE** { **RESPECT** | **IGNORE** }]
[**CHECKSUM** { **ON** | **OFF** }]
]

page-size :
1024 | **2048** | **4096** | **8192** | **16384** | **32768**

collation-label : *string*

key-spec:
[**ON**] **KEY** *key* [**ALGORITHM** { **'AES'** | **'AES_FIPS'** }]

Parameters

File name The file names (*db-file-name-string*, *log-file-name-string*, *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 being interpreted as new line characters (**\n**) or as hexadecimal numbers (**\x**), according to the rules for strings in SQL.

  It is safer to always escape the backslash character. For example,

  ```
  CREATE DATABASE 'c:\\sybase\\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 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, or *.mlg* for the mirror log.

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" [*ASA Database Administration Guide,* page 378]), and in data replication.

MIRROR clause 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, Adaptive Server Anywhere does not use a mirrored transaction log. If you do wish to use a transaction log mirror, this option allows you to provide a filename.

CASE clause For databases created with CASE RESPECT, all values are case sensitive in comparisons and string operations.

This option is provided for compatibility with the ISO/ANSI SQL standard. The default value for the option is CASE IGNORE; that is, all comparisons are case insensitive. User IDs and other identifiers in the database are case insensitive, even in case sensitive databases.

PAGE SIZE clause The page size for a database can be 1024, 2048, 4096, 8192, 16384, or 32768 bytes. The default page size is 2048 bytes. Large databases generally obtain performance benefits from a larger page size, but there can be additional overhead associated with large page sizes.

☞ For more information, see "Information utility options" [*ASA Database Administration Guide,* page 528].

For example,

```
CREATE DATABASE 'c:\\sybase\\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.

COLLATION clause The collation sequence used for all string comparisons in the database.

If you want to create a custom collation, use the Collation utility to create a file containing the collation. Once you have modified the collation and inserted it into the appropriate scripts, you can create the database and specify the new collation. You must change the collation label in the custom collation file. Otherwise, the you will not be able to create a database using the new collation since it conflicts with an existing collation.

☞ For more information on custom collation sequences, see "International

Languages and Character Sets" [*ASA Database Administration Guide,* page 319].

☞ For information on the Collation utility, see "The Collation utility" [*ASA Database Administration Guide,* page 503].

If you do not specify a collation, the default collation is used. Normal ASCII (binary) ordering is used for the lower 128 characters, subject to the case sensitivity setting (-c). For the upper 128 characters (also called the extended characters), any character that is an accented form of a letter in the lower 128 are sorted to the same position as the unaccented form. The determination of whether or not an extended character is an accented letter is based upon code page 850 (multilingual code page)

☞ For a list of the available collating sequence labels, see "Understanding collations" [*ASA Database Administration Guide,* page 335].

ENCRYPTED clause Encryption makes the data stored in your physical database file unreadable. There are two levels of encryption:

Simple encryption is equivalent to obfuscation. The data is unreadable, but someone with cryptographic expertise could decipher the data. Simple encryption is achieved by specifying the ENCRYPTED clause with no KEY clause.

Strong encryption is achieved through the use of a 128-bit AES algorithm and a security key. The data is unreadable and virtually undecipherable without the key. To create a strongly encrypted database, specify the ENCRYPTED clause with the KEY option. 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 upper and lower case, and includes numbers, letters, and special characters.

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.

You will require this key each time you want to start the database.

The following are invalid for database encryption keys:

♦ keys that begin with white space or single or double quotes

♦ keys that end with white space

♦ keys that contain semicolons

The ALGORITHM clause allows you to choose AES (the default) or AES_FIPS for the FIPS-approved algorithm. On Windows CE, only the AES algorithm is supported.

☞ For more information about strong database encryption, see "Strong encryption" [*SQL Anywhere Studio Security Guide*, page 15].

BLANK PADDING clause If you specify BLANK PADDING ON, trailing blanks are ignored in comparisons. For example, the two strings

```
'Smith'
'Smith    '
```

would be treated as equal in a database created with BLANK PADDING ON.

This option is provided for compatibility with the ISO/ANSI SQL standard, which is to ignore trailing blanks in comparisons. The default is that blanks are significant for comparisons (BLANK PADDING OFF).

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 (versions 4 and earlier of this software). These views conflict with the Sybase Adaptive Server Enterprise compatibility views dbo.syscolumns and dbo.sysindexes.

JCONNECT clause If you wish to use the Sybase jConnect JDBC driver to access system catalog information, you need to install jConnect support. Specify JCONNECT OFF if you wish to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information.

JAVA clause The default behavior is **JAVA OFF**.

To use Java in the database, you must install entries for the Sybase runtime Java classes into the system tables. Specifying **JAVA JDK '1.1.8'** or **JAVA JDK '1.3'** explicitly installs entries for the named version of the JDK. For JDK 1.1.8 the classes are held *java\1.1\classes.zip* under your SQL Anywhere directory. For JDK 1.3, they are held in *java\1.3\rt.jar*. The default classes are the JDK 1.3 classes.

Separately licensable option required
Java in the database is a separately licensable component and must be ordered before you can install it.

☞ To order this component, see "Separately-licensable components" [*Introducing SQL Anywhere Studio*, page 5].

PASSWORD CASE clause You can specify whether passwords are case sensitive in the database. The case sensitivity of passwords does not have to be the same as the database's case sensitivity setting for string comparisons. If you do not specify the case sensitivity of passwords, passwords follow the case sensitivity of the database. Extended characters used in passwords are case sensitive, regardless of the password case sensitivity setting.

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 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' )
```

☞ For more information, see "Validating a database using the dbvalid command-line utility" [*ASA Database Administration Guide*, page 605], "sa_validate system procedure" on page 838, or "VALIDATE CHECKSUM statement" on page 660.

Usage

Creates a database file with the supplied name and attributes.

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 server is running must have write permissions on the directories where files are created.

Not supported on Windows CE.

Side effects

An operating system file is created.

See also

- "ALTER DATABASE statement" on page 266
- "DROP DATABASE statement" on page 456
- "The Initialization utility" [*ASA Database Administration Guide*, page 530]
- "DatabaseKey connection parameter [DBKEY]" [*ASA Database Administration Guide*, page 187]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Adaptive Server Enterprise provides a CREATE DATABASE statement, but with different options.

Example

The following statement creates a database file named *mydb.db* in the *C:\* directory.

```
CREATE DATABASE 'C:\\mydb'
TRANSACTION LOG ON
CASE IGNORE
PAGE SIZE 1024
COLLATION '437'
ENCRYPTED OFF
BLANK PADDING OFF
JAVA JDK '1.3'
JCONNECT OFF
```

The following statement creates a database with no Sybase runtime Java classes. All database operations will execute normally, except for those involving Java classes or objects.

```
CREATE DATABASE 'C:\\nojava'
JAVA OFF
```

CREATE DBSPACE statement

Description

Use this statement to define a new database space and create the associated database file.

Syntax

CREATE DBSPACE *dbspace-name* **AS** *filename*

Parameters

dbspace-name An internal name for the database file. The *filename* parameter is the actual name of the database file, with a path where necessary.

filename A *filename* 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 *filename* is relative to the database server. When you are using the database server for NetWare, the *filename* should use a volume name (not a drive letter) when an absolute directory is specified.

Usage

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.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Automatic checkpoint.

See also

- "DROP statement" on page 454
- "Using additional dbspaces" [*ASA Database Administration Guide,* page 290]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

Create a dbspace called library to hold the LibraryBooks table and its indexes.

```
CREATE DBSPACE library
AS 'e:\\dbfiles\\library.db';
CREATE TABLE LibraryBooks (
  title char(100),
  author char(50),
  isbn char(30),
) IN library;
```

CREATE DECRYPTED FILE statement

Description

This statement decrypts strongly encrypted databases.

Syntax

CREATE DECRYPTED FILE *newfile*
FROM *oldfile* **KEY**
key

Parameters

FROM Lists the filename of the encrypted file.

KEY Lists the key required to access the encrypted file.

Usage

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 in order 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 -an option with either -ek or -ep. You can also use this method to change an existing encryption key.

Permissions

Must be a user with DBA authority.

Side effects

None.

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

Description

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 By default, domains allow NULLs unless the **allow_nulls_by_default** option is set to OFF. In this case, new domains by default do not allow NULLs. The nullability of a column created on a domain depends on the setting of the definition of the domain, not on the setting of the **allow_nulls_by_default** option when the column is referenced. Any explicit setting of NULL or NOT NULL in the column definition overrides the domain setting.

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.

Usage

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 in order to drop a domain.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "DROP statement" on page 454
- "SQL Data Types" on page 53

Standards and compatibility

- **SQL/92** Intermediate-level feature.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported by Adaptive Server Enterprise. Transact-SQL provides similar functionality using the sp_addtype system procedure and the CREATE DEFAULT and CREATE RULE statements.

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

Description

This statement encrypts strongly encrypted databases, transaction log files, or dbspaces.

Syntax

CREATE ENCRYPTED FILE *newfile*
FROM *oldfile*
KEY *key*
[**ALGORITHM** { **AES** | **AES_FIPS** }]

Parameters

FROM Lists the filename of the unencrypted file.

KEY Lists the key assigned to the encrypted file.

ALGORITHM Specifies the algorithm used to encrypt the file. If you do not specify an algorithm, AES is used by default.

Usage

This statement takes an unencrypted database, transaction log file, or dbspace and creates a new encrypted file. The original file must not be encrypted. The resulting file is an exact copy of the original file, except that it is encrypted using the specified algorithm and key.

If a database is encrypted using this statement, the corresponding transaction log file (and any dbspaces) must also be encrypted with the same algorithm and key in order 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 encrypt an existing database or change an existing encryption key by unloading and reloading the database using the -an option with either -ek or -ep. You can also use the CREATE ENCRYPTED FILE statement in conjunction with the CREATE DECRYPTED FILE statement to change an encryption key.

Permissions

Must be a user with DBA authority.

On Windows CE, only the AES algorithm is supported.

Side effects

None.

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 at 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'
```

Example 3

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'
```

Example 4

The following example changes the encryption key of the contacts database.

```
CREATE DECRYPTED FILE 'temp.db'
FROM 'contacts.db'
KEY 'oldkey'
del contacts.db
CREATE ENCRYPTED FILE 'contacts.db'
FROM 'temp.db'
KEY 'newkey'
del temp.db
```

CREATE EVENT statement

Description

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   | "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 "Choosing a system event" [*ASA Database Administration Guide,* page 307].

- **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 or on very early releases of Windows 95.

  The **TempDiskSpace** event type checks the amount of temporary disk space.

  If the appropriate event handlers have been defined (DBDiskSpace, LogDiskSpace, or TempDiskSpace), the 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 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.

  ☞ For more information, see "-fc server option" [*ASA Database Administration Guide,* page 139].

- **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 server checks for server activity every 30 seconds.

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 148.

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 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 wish 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.

Usage

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 server is idle, and so on.

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 wish to develop and 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.

When event handlers are triggered, the 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 150.

Permissions

Must have DBA authority.

Event handlers execute on a separate connection, with the permissions of the event owner. To execute with permissions 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.

The actions of an event handler are committed if no error is detected during execution, and rolled back if errors are detected.

See also

- "BEGIN statement" on page 312
- "ALTER EVENT statement" on page 273
- "COMMENT statement" on page 327

- ♦ "DROP statement" on page 454
- ♦ "TRIGGER EVENT statement" on page 641
- ♦ "EVENT_PARAMETER function [System]" on page 150
- ♦ "Choosing a system event" [*ASA Database Administration Guide*, page 307]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise.

Example

Instruct the database server to carry out an automatic backup to tape using the first tape drive on a Windows NT machine, every day at 1 am.

```
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 am and 6 pm.

```
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
```

☞ For more examples see "Defining trigger conditions for events" [*ASA Database Administration Guide*, page 308].

CREATE EXISTING TABLE statement

Description

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 filenames 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'
```

Usage

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 Adaptive Server Anywhere. This establishes the remote table as a visible entity to Adaptive Server Anywhere users. Adaptive Server 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 system table sysindexes. 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, Adaptive Server Anywhere derives the column list from the metadata it obtains from the remote table. If column-definitions are specified, Adaptive Server 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
- "Specifying proxy table locations" [*ASA SQL User's Guide*, page 614]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

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. Adaptive Server 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_employee for the employee table at the Adaptive Server Anywhere remote server asademo.

```
CREATE EXISTING TABLE rda_employee
AT ’asademo..DBA.employee’
```

CREATE EXTERNLOGIN statement

Description	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.

Usage

By default, Adaptive Server 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 SYSERVERS table. For more information, see "CREATE SERVER statement" on page 392.

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 the login-name and the DBA account 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 458

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

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

Description

Use this statement to create a new function in the database.

Syntax 1

CREATE FUNCTION [*owner*.]*function-name* **(** [*parameter*, . . .] **)**
RETURNS *data-type routine-characteristics*
{ *compound-statement*
  | **AS** *tsql-compound-statement*
  | *external-name* }

Syntax 2

CREATE FUNCTION [*owner*.]*procedure-name* **(** [*parameter*, . . .] **)**
RETURNS *data-type*
URL *url-string*
[**TYPE** { **'HTTP**[**:**{**GET**|**POST**}]**'** | **'SOAP**[**:**{**RPC**|**DOC**}]**'** }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]

url-string :
'{**HTTP**|**HTTPS**}**://**[*user*:*password*@]*hostname*[**:***port*][**/***path*]**'**

parameter :
[**IN**] *parameter-name data-type*

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 :
Windows95 | **WindowsNT** | **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.

compound-statement A set of SQL statements bracketed by BEGIN and END, and separated by semicolons. See "BEGIN statement" on page 312

tsql-compound-statement A batch of Transact-SQL statements. See "Transact-SQL batch overview" [*ASA SQL User's Guide*, page 498], and "CREATE PROCEDURE statement [T-SQL]" on page 383.

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" [*ASA SQL User's Guide*, page 710].

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 373.

ON EXCEPTION RESUME clause Use Transact-SQL -like error handling. For more information, see "CREATE PROCEDURE statement" on page 373.

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 a HTTP or SOAP web services client function. Specifies the URI of the web service. The optional username 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.

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 In order 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 may be placed in a file and the name of the file provided using the file key, or the whole certificate may be placed in a string, but not both. The following keys are available:

Key	Abbre-viation	Description
file		The filename 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 direced to an HTTPS server, or may be redirected from a non-secure to a secure server. A certificate value is not required when sent to an Adpative Server Anywhere database server that accepts HTTPS requests, unless a certificate parameter was used when the database server was started.

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 may specify a single port numbers, ranges of port numbers, or a combination of both; for example, CLIENTPORT ‘85,90-97’.

☞ For more information, see “ClientPort protocol option [CPORT]” [*ASA Database Administration Guide,* page 209].

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.

Usage

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.

Adaptive Server Anywhere treats all user-defined functions as idempotent unless they are declared NOT DETERMINISTIC. Idempotent functions return a consistent result for the same parameters and are free of side effects. That is, the 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 paramters 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.

External functions, including Java functions, must have DBA authority.

Side effects

Automatic commit.

See also

- ♦ "ALTER FUNCTION statement" on page 275
- ♦ "CREATE PROCEDURE statement" on page 373
- ♦ "DROP statement" on page 454
- ♦ "BEGIN statement" on page 312
- ♦ "CREATE PROCEDURE statement" on page 373
- ♦ "RETURN statement" on page 583
- ♦ "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- ♦ **SQL/92** Persistent Stored Module feature.
- ♦ **SQL/99** Persistent Stored Module feature.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise. The CREATE FUNCTION statement is supported by Microsoft SQL Server.

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 (emp_fname, emp_lname)
FROM employee
```

fullname (emp_fname, emp_lname)
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 dba.encrypt( IN name char(254) )
RETURNS VARCHAR
EXTERNAL NAME
  'Scramble.encrypt (Ljava/lang/String;)Ljava/lang/String;'
LANGUAGE JAVA
```

CREATE INDEX statement

Description

Use this statement to create an index on a specified table. Indexes can improve database performance.

Syntax

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*]

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.

☞ For more information, see "Starting the Index Consultant" [*ASA SQL User's Guide,* page 69], and "Index Consultant overview" [*ASA SQL User's Guide,* page 67].

CLUSTERED keyword The CLUSTERED attribute causes table rows to be stored in an approximate key order corresponding to the index. While the server makes an attempt to preserve key order, total clustering is not guaranteed.

If a clustered index exists, the LOAD TABLE statement inserts rows into the table in the order of the index key, and the INSERT statement attempts to put new rows on the same table page as the one containing adjacent rows, as defined by the key order.

☞ For more information, see "Using clustered indexes" [*ASA SQL User's Guide,* page 63].

UNIQUE keyword The UNIQUE attribute ensures that there will not be two rows in the table 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 on a table 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 column with a unique constraint, but not a unique index, because it can include multiple instances of NULL.

ASC | DESC option Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index will be 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 will be used only if the index was created with the same ascending and descending attributes.

function-name parameter The function-name parameter creates an index on a built-in function. 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 built-in function, along with any specified arguments. 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 column to be dropped. This form of CREATE INDEX cannot be used on declared temporary tables.

☞ For more information about computed columns, see "Working with computed columns" [*ASA SQL User's Guide*, page 49]

IN | ON clause By default, the index is placed in the same database file as its table. 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.

☞ For more information on limitations, see "Size and number limitations" [*ASA Database Administration Guide*, page 744].

Usage

The CREATE INDEX statement creates a sorted index on the specified columns of the named table. 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) or delete it (DROP INDEX).

You cannot create indexes on views.

♦ **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. The index name must be unique for each owner.

- **No indexes on views** Indexes cannot be created for views.
- **Index name space** The name of each index must be unique for a given table.
- **Exclusive table use** CREATE INDEX is prevented whenever the statement affects a table currently being used by another connection. CREATE INDEX can be time consuming and the server will not process requests referencing the same table while the statement is being processed.
- **Automatically created indexes** Adaptive Server Anywhere automatically creates indexes for primary keys and for 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 have either DBA authority or REFERENCES permission.

Side effects

Automatic commit. Creating an index on a built-in function also causes a checkpoint.

Column statistics are updated.

See also

- "DROP statement" on page 454
- "Indexes" [*ASA SQL User's Guide*, page 426]
- "Types of index" [*ASA SQL User's Guide*, page 432]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Adaptive Server Enterprise has a more complex CREATE INDEX statement than Adaptive Server Anywhere. While the Adaptive Server Enterprise syntax is permitted in Adaptive Server Anywhere, some clauses and keywords are ignored.
  - The full syntax for Adaptive Server Enterprise 11.5 is as follows:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]
INDEX index-name
ON [ [ database.]owner.]table_name
     (column_name [, column_name], . . . )
 [ WITH {
   {  FILLFACTOR | MAX_ROWS_PER_PAGE } = x,
      CONSUMERS = x,
   . . . IGNORE_DUP_KEY,
   . . . SORTED_DATA,
      [ IGNORE_DUP_ROW | ALLOW_DUP_ROW ]
   } ]
 [ ON segment_name ]
```

Adaptive Server Anywhere allows, by ignoring, the following keywords:

- FILLFACTOR
- IGNORE_DUP_KEY
- SORTED_DATA
- IGNORE_DUP_ROW
- ALLOW_DUP_ROW

Physical placement of an index is carried out differently in Adaptive Server Enterprise and Adaptive Server Anywhere. The **ON** *segment-name* clause is supported in Adaptive Server Anywhere, but *segment-name* refers to a dbspace.

Unique indexes in Adaptive Server Anywhere permit entries that contain NULL, and are otherwise identical. Unique indexes in Adaptive Server Enterprise do not permit entries that contain NULL and are otherwise identical.

Index names must be unique on a given table for both Adaptive Server Anywhere and Enterprise.

Example

Create a two-column index on the employee table.

```
CREATE INDEX employee_name_index
ON employee
( emp_lname, emp_fname )
```

Create an index on the sales_order_items table for the prod_id column.

```
CREATE INDEX item_prod
ON sales_order_items
( prod_id )
```

Use the SORTKEY function to create an index on the description column of the product 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 product (
 SORTKEY( description, ’rusdict’ )
 AS desc_ru )
```

CREATE MESSAGE statement [T-SQL]

Description

Use this statement to add a user-defined message to the SYSUSERMESSAGES 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**.

Usage

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 454.

Permissions

Must have RESOURCE authority

Side effects

Automatic commit.

See also

- “PRINT statement [T-SQL]” on page 566
- “RAISERROR statement [T-SQL]” on page 569

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** The functionality of CREATE MESSAGE is provided by the **sp_addmessage** procedure in Adaptive Server Enterprise.

CREATE PROCEDURE statement

Description

Use this statement to create a procedure in the database.

Syntax 1

CREATE 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**]
}

Syntax 2

CREATE PROCEDURE [*owner*.]*procedure-name* **(** [*parameter*, . . .] **)**
compound-statement

Syntax 3

CREATE PROCEDURE [*owner*.]*procedure-name* **(** [*parameter*, . . .] **)**
URL *url-string*
[**TYPE** { **'HTTP**[**:**{**GET**|**POST**}]**'** | **'SOAP**[**:**{**RPC**|**DOC**}]**'** }]
[**NAMESPACE** *namespace-string*]
[**CERTIFICATE** *certificate-string*]
[**CLIENTPORT** *clientport-string*]
[**PROXY** *proxy-string*]

url-string :
'{**HTTP**|**HTTPS**}**://**[*user***:***password*@]*hostname*[**:***port*][**/***path*]**'**

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 :
Windows95 | **WindowsNT** | **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 PROCEDURE clause 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 53). Parameters can be prefixed them with one of the keywords IN, OUT, or INOUT. If you do not specify one of these values, parameters are INOUT parameters 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.

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 53.

☞ For more information on returning result sets from procedures, see "Returning results from procedures" [*ASA SQL User's Guide*, page 686].

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 emp_fname
      FROM employee
   ELSE
      SELECT emp_lname,emp_fname
      FROM employee
   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, in order 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 redescribing the result set after a cursor is open.

In order 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.

☞ For more information, see "ON_TSQL_ERROR option [compatibility]" [*ASA Database Administration Guide,* page 675].

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" [*ASA SQL User's Guide,* page 710].

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 filenames and extensions to be used in the database and owner fields.

For example, the following statement creates a proxy procedure (remotewho) that calls the dbo.sp_who procedure on the master database of the bostonase server:

```
CREATE PROCEDURE remotewho ()
AT 'bostonase.master.dbo.sp_who
```

Remote procedures can return only up to 254 characters in output variables.

☞ For information on remote servers, see "CREATE SERVER statement" on page 392. For information on using remote procedures, see "Using remote procedure calls (RPCs)" [*ASA SQL User's Guide*, page 623].

DYNAMIC RESULT SETS clause This clause 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.

URL clause For use only when defining a HTTP or SOAP web services client function. Specifies the URI of the web service. The optional username 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.

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 In order 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 may be placed in a file and the name of the file provided using the file key, or the whole certificate may be placed in a string, but not both. The following keys are available:

Key	Abbre-viation	Description
file		The filename of the certificate.
certifi-cate	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 direced to an HTTPS server, or may be redirected from a non-secure to a secure server. A certificate value is not required when sent to an Adpative Server Anywhere database server that accepts HTTPS requests, unless a certificate parameter was used when the database server was started.

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 may specify a single port numbers, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'.

☞ For more information, see "ClientPort protocol option [CPORT]" [*ASA Database Administration Guide,* page 209].

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.

EXTERNAL NAME LANGUAGE JAVA clause A procedure that uses EXTERNAL NAME with a LANGUAGE JAVA clause is a wrapper around a Java method.

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;**.

A Java method signature is a compact character representation of the types of the parameters and the type of the return value. It is an error to put a space before the signature.

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'
```

☞ For more information, see "Returning result sets from Java methods" [*ASA Programming Guide*, page 94].

Usage

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.

Parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the paramters 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.

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 paramters 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.

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 312
- "CALL statement" on page 318
- "CREATE FUNCTION statement" on page 362
- "CREATE PROCEDURE statement [T-SQL]" on page 383
- "ALTER PROCEDURE statement" on page 278
- "DROP statement" on page 454
- "EXECUTE IMMEDIATE statement [SP]" on page 475
- "GRANT statement" on page 503
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** The Transact-SQL CREATE PROCEDURE statement is different.
- **SQLJ** The syntax extensions for Java result sets are as specified in the proposed SQLJ1 standard.

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 "DBA"."product"
   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 company_name,
          CAST(sum(sales_order_items.quantity *
          product.unit_price) AS INTEGER) VALUE
      FROM customer
      LEFT OUTER JOIN sales_order
      LEFT OUTER JOIN sales_order_items
      LEFT OUTER JOIN product
      GROUP BY company_name;
   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]

Description

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 Adaptive Server 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.

Usage

The following differences between Transact-SQL and Adaptive Server 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 Adaptive Server 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 employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = @deptname
   AND department.dept_id = employee.dept_id
```

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 employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = deptname
   AND department.dept_id = employee.dept_id
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 373

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** 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. Adaptive Server 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

Description

Use this statement to create a publication. In MobiLink, a publication identifies synchronized data in UltraLite or Adaptive Server Anywhere remote databases.

In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax

CREATE PUBLICATION [*owner*.]*publication-name*
(TABLE *article-description*, . . . **)**

owner, *publication-name* : *identifier*

article-description :
  table-name [**(** *column-name*, . . . **)**]
[**WHERE** *search-condition*]
[**SUBSCRIBE BY** *expression*]

Parameters

article-description Publications are built from articles. Each article is a table or part of a table. An article may be a vertical partition of a table (a subset of the table's columns), a horizontal partition (a subset of the table's rows) or a vertical and horizontal partition.

WHERE clause The WHERE clause is a way of defining the subset of rows of a table to be included in an article. It is useful if the same subset is to be received by all subscribers to the publication.

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. This clause is ignored during MobiLink synchronization.

You can combine WHERE and SUBSCRIBE BY clauses in an article definition, but the SUBSCRIBE BY clause is used only by SQL Remote.

Usage

This statement is applicable only to MobiLink and SQL Remote.

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 Adaptive Server Anywhere remote databases, and are optional in UltraLite databases. These publications and the subscriptions to them determine which data will be uploaded to the MobiLink synchronization server. You can construct a remote database by creating publications and subscriptions directly. Alternatively, you can create publications and subscriptions in an Adaptive Server Anywhere reference database, which acts as a template for the remote databases, and then construct the remote databases using the MobiLink extraction utility.

You set options for a MobiLink publication with the ADD OPTION clause in the ALTER SYNCHRONIZATION SUBSCRIPTION statement or CREATE SYNCHRONIZATION SUBSCRIPTION statement.

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" on page 280
- "DROP PUBLICATION statement" on page 459
- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 290
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 402
- "sp_create_publication procedure" [*SQL Remote User's Guide*, page 386]
- Adaptive Server Anywhere MobiLink clients: "Publishing data" [*MobiLink Clients*, page 64]
- UltraLite MobiLink clients: " Choosing data to synchronize" [*MobiLink Clients*, page 280]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Example

The following statement publishes all columns and rows of two tables.

```
CREATE PUBLICATION pub_contact (
   TABLE contact,
   TABLE company
)
```

The following statement publishes only some columns of one table.

```
CREATE PUBLICATION pub_customer (
   TABLE customer ( id, company_name, city )
)
```

The following statement publishes only the active customer rows by including a WHERE clause that tests the status column of the customer table.

```
CREATE PUBLICATION pub_customer (
   TABLE customer ( id, company_name, city, state )
   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 customer ( id, company_name, 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
```

CREATE REMOTE MESSAGE TYPE statement [SQL Remote]

Description

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.

Usage

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 e-mail system, the address string must be a valid e-mail 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.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 510
- "GRANT REMOTE statement [SQL Remote]" on page 511
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 508
- "DROP REMOTE MESSAGE TYPE statement [SQL Remote]" on page 460
- "sp_remote_type procedure" [*SQL Remote User's Guide*, page 427]
- "Using message types" [*SQL Remote User's Guide*, page 210]

Standards and compatibility

- **SQL/92** Vendor extension.

- **SQL/99** 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

Description	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
 ], . . .
```

Usage

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 407
- “CREATE VIEW statement” on page 428
- “GRANT statement” on page 503

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.

- ♦ **Sybase** Adaptive Server 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

Description

Use this statement to add a server to the SYSSERVERS system table.

Syntax

CREATE SERVER *server-name*
CLASS *'server-class'*
USING *'connection-info'*
[**READ ONLY**]

server-class :
 ASAJDBC | **ASEJDBC**
| **ASAODBC** | **ASEODBC**
| **DB2ODBC** | **MSSODBC**
| **ORAODBC** | **ODBC**

connection-info :
 { *machine-name*:*port-number* [/*dbname*] | *data-source-name* | *asa-connection-string* }

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 asajdbc class is supported.

USING clause The USING clause supplies a connection string for the 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** is the machine the remote server runs on
- **portnumber** is the TCP/IP port number the remote server listens on. The default port number for Adaptive Server Anywhere is 2638.
- **dbname** For Adaptive Server 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 Adaptive Server Anywhere remote servers (ASAJDBC or ASAODBC server classes), the *connection-info* parameter can be any valid Adaptive Server Anywhere connection string. You can use any Adaptive Server 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 Adaptive Server Anywhere connection strings, see "Connection parameters" [*ASA Database Administration Guide*, page 176].

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=/opt/sybase/SYBSsa9/drivers/lib/libodbc.so;dsn=my_
         dsn'
```

READ ONLY The READ ONLY clause specifies that the remote server is a read-only data source. Any update request is rejected by Adaptive Server Anywhere.

Usage

The CREATE SERVER statement defines a remote server from the Adaptive Server Anywhere catalogs.

☞ For more information on server classes and how to configure a server, see "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635].

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 283
- "DROP SERVER statement" on page 461
- "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

The following example creates an Adaptive Server Anywhere remote server named testasa, located on the machine *apple* and listening on port number 2638, use:

```
CREATE SERVER testasa
CLASS 'asajdbc'
USING 'apple:2638'
```

The following example creates a remote server for the JDBC-based Adaptive Server named ase_prod. Its machine name is banana and 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’
```

CREATE SERVICE statement

Description

Use this statement to permit a database server to act as a web server.

Syntax 1

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**FORMAT** { **'DNET'** | **'CONCRETE'** | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

Syntax 3

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 may be any sequence of alpha-numeric 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.

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 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 may 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 unsecure 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 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.

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. The statement is mandatory for SOAP services, and ignored for DISH 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.

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 ASADataset 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 are 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.

Usage

The create service statement causes the database server to act as a web server. A new entry is created in the SYSWEBSERVICE system table.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER SERVICE statement" on page 286, "DROP SERVICE statement" on page 462

Examples

To quickly set up a web server, start a database server with the -xs switch, then execute the following statement:

```
CREATE SERVICE tables TYPE 'XML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTABLE
```

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;
```

CREATE STATISTICS statement

Description

This statement should be used only in rare circumstances. It explicitly recreates the statistics that are used by the optimizer.

Syntax

CREATE STATISTICS *table-name* [**(** *column-list* **)**]

Usage

This statement recreates the statistics that Adaptive Server Anywhere uses to optimize database queries. These statistics analyze the distribution of data in the database for the specified table. The process of running CREATE STATISTICS performs a sequential scan of the entire table.

In rare circumstances, when your database queries are very variable, and when data distribution is not uniform or the data is changing frequently, you may improve performance by running CREATE STATISTICS against a table or column. This causes a sequential scan of the table or column, using an index if possible.

CREATE STATISTICS overwrites existing statistics. You do not need to drop statistics before executing it.

CREATE STATISTICS creates histograms for the specified table, regardless of the size of the table (by default, histograms are created for tables with five or more rows).

You can also create statistics using the LOAD TABLE statement.

Permissions

Must have DBA authority.

Side effects

Query plans will probably change.

See also

- "DROP STATISTICS statement" on page 464
- "ALTER DATABASE statement" on page 266
- "LOAD TABLE statement" on page 538

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

CREATE SUBSCRIPTION statement [SQL Remote]

Description
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*

Usage
In a SQL Remote installation, data is organized into **publications** for replication. In order 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.

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 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.

Permissions
Must have DBA authority.

Side effects
Automatic commit.

See also

- ♦ "DROP SUBSCRIPTION statement [SQL Remote]" on page 465
- ♦ "GRANT REMOTE statement [SQL Remote]" on page 511
- ♦ "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 639
- ♦ "START SUBSCRIPTION statement [SQL Remote]" on page 628
- ♦ "sp_subscription procedure" [*SQL Remote User's Guide*, page 433]

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]

Description

Use this statement in an Adaptive Server Anywhere remote database to subscribe a MobiLink user to a publication.

Syntax

CREATE SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, . . .]
[**TYPE** *protocol-type*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, . . .]

ml_username: *identifier*

protocol-type: **http** | **https** | **https_fips** | **tcpip** | **ActiveSync**

protocol-options: *string*

value: *string* | *integer*

Parameters

TO clause Specify the name of a publication.

FOR clause Specify one or more MobiLink user names.

ml_username is a name identifying a remote database. This name must be unique.

☞ For more information about synchronization user names, see "About MobiLink users" [*MobiLink Clients,* page 10].

Omit the FOR clause to set the protcol 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 for extended options and connection parameters" [*MobiLink Clients,* page 73].

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 Clients,* page 111].

ADDRESS clause This clause specifies network protocol options such as the location of the MobiLink synchronization server. Multiple options must be separated with semi-colons.

☞ For a complete list of protocol options, see "CommunicationAddress (adr) extended option" [*MobiLink Clients,* page 106].

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, and are overridden by any extended options set for a synchronization user.

☞ For information about how dbmlsync processes options that are specified in different locations, see "Priority order for extended options and connection parameters" [*MobiLink Clients,* page 73].

☞ For a complete list of options, see "dbmlsync extended options" [*MobiLink Clients,* page 105].

Usage

Use this statement to create a synchronization subscription within a MobiLink remote or reference database.

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" on page 385
- "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 404

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Examples

Create a default subscription, which contains default subscription values, for the sales publication (by omitting the FOR clause). Indicate the address of the MobiLink synchronization server and specify that only the Certicom root certificate is to be trusted.

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   ADDRESS 'host=test.internal;port=2439;
      security=ecc_tls'
   OPTION memory='2m';
```

Subscribe MobiLink user ml_user1 to the sales publication. Set the memory option to 3 Mb, rather than the value specified in the default publication.

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR ml_user1
   OPTION memory='3m';
```

CREATE SYNCHRONIZATION USER statement [MobiLink]

Description

Use this statement in an Adaptive Server Anywhere remote database to create a synchronization user.

Syntax

CREATE SYNCHRONIZATION USER *ml_username*
[**TYPE** *protocol-type*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, . . .]

ml_username: *identifier*

protocol-type: **tcpip** | **http** | **https** | **https_fips** | **ActiveSync**

protocol-options: *string*

value: *string* | *integer*

Parameters

ml_username A name identifying a remote database. This name must be unique.

☞ For more information about synchronization user names, see "About MobiLink users" [*MobiLink Clients*, page 10].

TYPE clause This clause specifies the communication protocol to use for synchronization. The options are **tcpip**, **http**, **https**, **https_fips**, and **ActiveSync**. The default protocol is **tcpip**.

☞ For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink Clients*, page 111].

ADDRESS clause This clause specifies *protocol-options* in the form *keyword*=*value*, separated by semi-colons. Which settings you supply depends on the communication protocol you are using (TCPIP, HTTP, HTTPS, or ActiveSync).

☞ For a complete list of protocol options, see "CommunicationAddress (adr) extended option" [*MobiLink Clients*, page 106].

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 "dbmlsync extended options" [*MobiLink Clients*, page 105].

Description

The *protocol-type*, *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 for extended options and connection parameters" [*MobiLink Clients*, page 73].

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- ♦ "ALTER SYNCHRONIZATION USER statement [MobiLink]" on page 292
- ♦ "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 402
- ♦ "CREATE PUBLICATION statement" on page 385
- ♦ "MobiLink Transport-Layer Security" [*MobiLink Administration Guide*, page 165]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Supported by Open Client/Open Server.

Examples

The following example creates a user named SSinger, who synchronizes over TCP/IP with a server machine 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'
```

The following creates a synchronization user called factory014 that will cause dbmlsync to hover and synchronize via Certicom-encrypted TCP/IP at a random time in every 8-hour interval. The randomness helps prevent performance degradation at the MobiLink server due to multiple simultaneous synchronizations:

```
CREATE SYNCHRONIZATION USER factory014
TYPE tcpip
ADDRESS 'host=mycompany.manufacturing.mobilink1;security=certico
         m_tls(certificate=mycompany_mobilink.crt;certificate_
         password=thepassword)'
OPTION Schedule='EVERY:08:00'
```

The following creates a synchronization user called sales5322 that will be used to synchronize with HTTP. In this example, the MobiLink synchronization server runs behind the corporate firewall, and synchronization requests are redirected to it using the Redirector (a reverse proxy to an NSAPI Web server).

```
CREATE SYNCHRONIZATION USER sales5322
TYPE https
ADDRESS 'host=www.mycompany.com;port=80;url_
         suffix=mlredirect/ml/'
```

CREATE TABLE statement

Description

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*]
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**
  | **NOT TRANSACTIONAL**]
[**AT** *location-string*]

column-definition :
column-name data-type [[**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
  | **PRIMARY KEY** [**CLUSTERED**]
  | **REFERENCES** *table-name*
      [**(** *column-name* **)**] [*actions*] [**CLUSTERED**]
}
| [**CONSTRAINT** *constraint-name*] **CHECK (** *condition* **)**
| **COMPUTE (** *expression* **)**

table-constraint :
[**CONSTRAINT** *constraint-name*] {
    UNIQUE (*column-name*, . . . **)**
  | **PRIMARY KEY** [**CLUSTERED**] **(** *column-name*, . . . **)**
  | **CHECK (** *condition* **)**
  | *foreign-key-constraint*
}

foreign-key-constraint :
[**NOT NULL**] **FOREIGN KEY** [*role-name*] [**(***column-name*, . . . **)**]
REFERENCES *table-name* [**(***column-name*, . . . **)**] [**CLUSTERED**]
[*actions*] [**CHECK ON COMMIT**]

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 344.

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.

NOT TRANSACTIONAL The NOT TRANSACTIONAL clause is allowed only for temporary tables. A table created using NOT TRANSACTIONAL 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 may be useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

AT clause Create a remote table on a different server specified by *location-string* and also 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 filenames 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 TABLE a1
AT 'access;d:\mydbfile.mdb;;a1'
```

☞ For information on remote servers, see "CREATE SERVER statement" on page 392. For information on proxy tables, see "CREATE EXISTING TABLE statement" on page 357 and "Specifying proxy table locations" [*ASA SQL User's Guide*, page 614].

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 server supports primary keys.

The COMPUTE clause is ignored for remote tables.

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

♦ **data-type** For information on data types, see "SQL Data Types" on page 53.

♦ **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 33.

  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 is specified, it is equivalent to DEFAULT NULL.

  Some of the defaults require more description:

  - **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, it is used; if the specified value is larger than the current maximum value for the column, that value will be 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 row number less then the maximum, subsequent rows without explicit assignment are still automatically incremented with a value of one greater than the previous maximum.

The next value to be used for each column is stored as an integer. Using values greater than (2^{31} – 1) may cause wraparound to incorrect values, and AUTOINCREMENT should not be used in such cases.

You can find the most recently inserted value of the column by inspecting the @@identity global variable.

The identity column is a Transact-SQL-compatible alternative to using the AUTOINCREMENT default. In Adaptive Server Anywhere, the identity column is implemented as AUTOINCREMENT default. For information, see "The special IDENTITY column" [*ASA SQL User's Guide,* page 490].

- **GLOBAL AUTOINCREMENT** This default is intended for use when multiple databases will be used in a SQL Remote replication or MobiLink synchronization environment.

  This default 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. Adaptive Server 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 may 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 our 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 $n\ p + 1$ to $(n + 1)\ p$, 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 $(n + 1)\ p$, the next default value will be one greater than the previous largest value in column. If the column contains no values, the first default value is $n\ p + 1$. Default column values are not affected by values in the column outside of the current partition; that is, by numbers less than $pn + 1$ or greater than $p(n + 1)$. Such values may be present if they have been replicated from another database via MobiLink synchronization.

Because the public option **Global_database_id** cannot be set to negative values, 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. Should null values not be permitted, attempting to insert the row causes an error. This situation arises, for example, if the column is contained in the table's primary key.

Null default values are also generated when the supply of values within the partition has been exhausted. 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. Attempting to insert the null value causes an error if the column does not permit nulls. To detect that the supply of unused values is low and handle this condition, create an event of type **GlobalAutoincrement**.

You cannot use DEFAULT GLOBAL AUTOINCREMENT in databases created with version 6 or earlier software, even if they have been upgraded.

- **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.

- **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" [*ASA SQL User's Guide,* page 488].

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.

☞ For more information, see "DEFAULT_TIMESTAMP_INCREMENT option [database]" [*ASA Database Administration Guide,* page 650].

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

For more information, see "TRUNCATE_TIMESTAMP_VALUES option [database]" [*ASA Database Administration Guide,* page 697].

The global variable **@@dbts** returns a TIMESTAMP value representing the last value generated for a column using DEFAULT TIMESTAMP. For more information, see "Global variables" on page 40.

- **string** For more information, see "Strings" on page 9.
- **global-variable** For more information, see "Global variables" on page 40.

♦ **column-constraint** A column constraint restricts the values the column can hold.

table-constraint A table constraint restricts the values that one or more columns in the table can hold.

Constraints 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. Column constraints are abbreviations for the corresponding table constraints.

♦ For example, the following statements are equivalent:

```
CREATE TABLE Product (
   product_num INTEGER UNIQUE
)
CREATE TABLE Product (
   product_num INTEGER,
   UNIQUE ( product_num )
)
```

Column constraints are normally used unless the constraint references more than one column in the table. In these cases, a table constraint must be used.

Constraints include the following:

- **CHECK** 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 constraint. If an INSERT or UPDATE statement would cause a row to violate a constraint, the operation is not permitted and the effects of the statement are undone.

  The change is rejected only if a constraint condition evaluates to FALSE, the change is allowed if a constraint condition evaluates to TRUE or UNKNOWN.

  ☞ For more information about TRUE, FALSE, and UNKNOWN conditions, see "NULL value" on page 49 and "Search conditions" on page 23.

- **COMPUTE** The COMPUTE constraint is a column constraint only. When a column is created using a COMPUTE constraint, 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.

  Any UPDATE statement that attempts to change the value of a computed column does fire any triggers associated with the column.

- **UNIQUE** 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 may have more than one unique constraint.

  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 column with a unique constraint, but not a unique index, because it can include multiple instances of NULL.

  ☞ For information about unique indexes, see "CREATE INDEX statement" on page 368.

- **PRIMARY KEY** This is the same as a unique constraint, except that a table can have only one primary key constraint. The primary key usually identifies the best identifier for a row. For example, the customer number might be the primary key for the customer table.

  Columns included in primary keys cannot allow NULL. Each row in the table has a unique primary key value. A table can have only one PRIMARY KEY.

The order of the columns in a primary key is the order in which the columns were created in the table, not the order in which they are listed when the primary key is created.

☞ For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*ASA SQL User's Guide*, page 63].

♦ **Foreign key** A foreign key constraint can be implemented using a REFERENCES column constraint (single column only) or a FOREIGN KEY table constraint. It restricts the values for a set of columns to match the values in a primary key or, less commonly, 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 customer table.

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 references the primary key of the primary table.

☞ For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*ASA SQL User's Guide*, page 63].

If you do not explicitly define a foreign key column, it 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.

If foreign key column names are specified, then primary key column names must also be specified, and the column names are paired according to position in the lists. 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.

If at least one value in a multi-column foreign key is NULL, there is no restriction on the values that can be held in other columns of the key.

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** Disallow NULL 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** 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.

♦ **action** The referential integrity action defines the action to be taken to maintain foreign key relationships in the database. Whenever a primary key value is changed or deleted from a database table, there may be corresponding foreign key values in other tables that should be modified in some way. You can specify either an ON UPDATE clause, an ON DELETE clause, or both, followed by one of the following actions:

- **CASCADE** When used with ON UPDATE, updates the corresponding foreign keys to match the new primary key value. When used with ON DELETE, deletes the rows from the foreign table that match the deleted primary key.
- **SET NULL** Sets to NULL all the foreign key values that correspond to the updated or deleted primary key.
- **SET DEFAULT** Sets foreign key values that match the updated or deleted primary key value to values specified on the DEFAULT clause of each foreign key column.
- **RESTRICT** Generates an error if an attempt is made to update or delete a primary key value while there are corresponding foreign keys elsewhere in the database. RESTRICT is the default action.

♦ **CHECK ON COMMIT** The CHECK ON COMMIT clause 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 clause does not delay CASCADE, SET NULL, or SET DEFAULT actions.

If you use CHECK ON COMMIT with out specifying any actions, then RESTRICT is implied as an action for UPDATE and DELETE.

PCTFREE 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, 200 bytes are reserved in each page.

The value for PCTFREE is stored in the SYSATTRIBUTE system table.

☞ For more information, see "SYSATTRIBUTE system table" on page 678.

Usage

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.

The definition of a temporary table exists in the database, like that of a base table, and remains in the database until it is explicitly removed by a DROP TABLE statement. The rows in a temporary table are visible only to the connection that inserted the rows. Multiple connections from the same or different applications can use the same temporary table at the same time, and each connection will see only its own rows. The rows of a temporary table for a connection are deleted when the connection ends.

Permissions

Must have RESOURCE authority.

Must have DBA authority to create a table for another user.

The AT clause to create proxy tables is not supported on Windows CE.

Side effects

Automatic commit.

See also

- "ALTER TABLE statement" on page 294
- "CREATE DBSPACE statement" on page 344
- "CREATE EXISTING TABLE statement" on page 357
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 443
- "DROP statement" on page 454
- "Special values" on page 33
- "SQL Data Types" on page 53
- "Creating tables" [*ASA SQL User's Guide*, page 40]
- "ALLOW_NULLS_BY_DEFAULT option [compatibility]" [*ASA Database Administration Guide*, page 633]

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.

  The following are vendor extensions:

  - The { **IN** | **ON** } *dbspace-name* clause.
  - The **ON COMMIT** clause
  - Some of the default values.

- ♦ **Sybase** Supported by Adaptive Server Enterprise, with some differences.
  - **Temporary tables** You can create a temporary table by preceding the table name in a CREATE TABLE statement with a pound sign (#). In Adaptive Server Anywhere, these are declared temporary tables, which are available only in the current connection. For information, see "DECLARE LOCAL TEMPORARY TABLE statement" on page 443.
  - **Physical placement** Physical placement of a table is carried out differently in Adaptive Server Anywhere and in Adaptive Server Enterprise. The **ON** *segment-name* clause supported by Adaptive Server Enterprise is supported in Adaptive Server Anywhere, but *segment-name* refers to a dbspace name.
  - **Constraints** Adaptive Server Anywhere does not support named constraints or named defaults, but does support domains, which allow constraint and default definitions to be encapsulated in the data type definition. It also supports explicit defaults and CHECK conditions in the CREATE TABLE statement.
  - **NULL default** By default, columns in Adaptive Server Enterprise default to NOT NULL, whereas in Adaptive Server Anywhere the default setting is NULL. This setting can be controlled using the ALLOW_NULLS_BY_DEFAULT database option. You should explicitly specify NULL or NOT NULL to make your data definition statements transferable between Adaptive Server Anywhere and Adaptive Server Enterprise.

    ☞ For more information, see "ALLOW_NULLS_BY_DEFAULT option [compatibility]" [*ASA Database Administration Guide,* page 633].

Example

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

Description

Use this statement to create a trigger on a table.

Syntax

CREATE TRIGGER *trigger-name trigger-time* { *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* **)**]
  compound-statement

trigger-time : **BEFORE** | **AFTER** | **RESOLVE**

trigger-event-list : *trigger-event* [, *trigger-event*]

trigger-event :
DELETE | **INSERT** | **UPDATE**

Parameters

Trigger events Triggers can be fired by one or more of the following events:

- **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.

You may 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.

☞ For more information, see "IF statement" on page 518.

trigger-time Row-level triggers can be defined to execute BEFORE or AFTER the insert, update, or delete. Statement-level triggers execute AFTER the statement. The RESOLVE trigger time is for use with SQL Remote: it fires before row-level UPDATE or UPDATE OF column-lists only.

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.

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.

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.

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.

Usage

The CREATE TRIGGER statement creates a trigger associated with a table in the database, and stores the trigger in the database.

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 312
- "CREATE PROCEDURE statement" on page 373
- "CREATE TRIGGER statement [T-SQL]" on page 426
- "DROP statement" on page 454
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- **SQL/92** Persistent stored module feature. Some clauses are vendor extensions.
- **SQL/99** Persistent Stored Module feature. Some clauses are vendor extensions.
- **Sybase** This syntax is different to that supported by Adaptive Server Enterprise.

Example

The first example creates a row-level trigger. When a new department head is appointed, update the **manager_id** column for employees in that department.

```
CREATE TRIGGER tr_manager
BEFORE UPDATE OF dept_head_id
ON department
REFERENCING OLD AS old_dept NEW AS new_dept
FOR EACH ROW
BEGIN
   UPDATE employee
   SET employee.manager_id=new_dept.dept_head_id
   WHERE employee.dept_id=old_dept.dept_id
END
```

The next example, which is more complex, deals with a statement-level trigger. First, create a table as follows:

```
CREATE TABLE "DBA"."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 DBA."insert-st" AFTER INSERT ORDER 4 ON
DBA.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
```

CREATE TRIGGER statement [SQL Remote]

Description

Use this statement to create a new trigger in the database. One form of trigger is designed specifically for use by SQL Remote.

Syntax

```
CREATE TRIGGER trigger-name trigger-time
trigger-event, . . .
[ 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 ) ]
[ IF UPDATE ( column-name ) THEN
[ { AND | OR } UPDATE ( column-name ) ] . . . ]
  compound-statement
[ ELSEIF UPDATE ( column-name ) THEN
[ { AND | OR } UPDATE ( column-name ) ] . . .
   compound-statement
END IF ] ]
```

```
trigger-time:
BEFORE | AFTER | RESOLVE
```

```
trigger-event:
DELETE | INSERT | UPDATE
| UPDATE OF column-name [, column-name, . . . ]
```

Parameters

trigger-time Row-level triggers can be defined to execute BEFORE or AFTER the insert, update, or delete. Statement-level triggers execute AFTER the statement. The RESOLVE trigger time is for use with SQL Remote: it fires before row-level UPDATE or UPDATE OF column-lists only.

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.

Trigger events Triggers can be fired by one or more of the following events:

- **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.

Usage	Anywhere.
Permissions	Must have RESOURCE authority and have ALTER permissions on the table, or must 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	♦ "UPDATE statement" on page 650

Description

The CREATE TRIGGER statement creates a trigger associated with a table in the database and stores the trigger in the database.

BEFORE UPDATE triggers fire any time an update occurs on a row, regardless of whether or not the new value differs from the old value. AFTER UPDATE triggers will fire only if the new value is different from the old value.

Row and statement-level triggers

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.

Row-level triggers can be defined to execute BEFORE or AFTER the insert, update, or delete. Statement-level triggers execute AFTER the statement. The RESOLVE trigger time is for use with SQL Remote; it fires before row-level UPDATE or UPDATE OF column-lists only.

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 of firing

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.

Referencing deleted and inserted values

The REFERENCING OLD and REFERENCING NEW clauses allow you to refer to the deleted and inserted rows. For the purposes of this clause, an UPDATE is treated as a delete followed by an insert.

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.

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 WHEN clause causes the trigger to fire only for rows where the search-condition evaluates to true.

Updating values with the same value

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.

Example

♦ When a new department head is appointed, update the **manager_id** column for employees in that department.

```
CREATE TRIGGER
tr_manager BEFORE UPDATE OF dept_head_id ON department
REFERENCING OLD AS old_dept
NEW AS new_dept
FOR EACH ROW
BEGIN
   UPDATE employee
   SET employee.manager_id=new_dept.dept_head_id
   WHERE employee.dept_id=old_dept.dept_id
END
```

CREATE TRIGGER statement [T-SQL]

Description	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

Usage

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 419

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/92** Transact-SQL extension.
- **Sybase** Anywhere supports a subset of the Adaptive Server Enterprise syntax.

CREATE VARIABLE statement

Description

Use this statement to create a SQL variable.

Syntax

CREATE VARIABLE *identifier data-type*

Usage

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. If a column name exists with the same name as the variable, the variable value is used.

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" [*ASA SQL User's Guide*, page 680]).

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 312
- "SQL Data Types" on page 53
- "DROP VARIABLE statement" on page 468
- "SET statement" on page 604

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

For an example, see "SET statement" on page 604

CREATE VIEW statement

Description

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. In order 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 597). All items in the select list must have unique names.

AS clause The SELECT statement on which the view is based must not have an ORDER BY clause on it. It may have a GROUP BY clause and may be a UNION. The SELECT statement must not refer to local temporary tables.

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.

Usage

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 operation. An update to the view causes the underlying table(s) to be updated.

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 454
- ♦ “CREATE TABLE statement” on page 407

Standards and compatibility

- ♦ **SQL/92** Entry-level feature.
- ♦ **SQL/99** Core feature.
- ♦ **Sybase** Supported by Adaptive Server Enterprise.

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 male_employee
AS SELECT *
FROM Employee
WHERE Sex = ’M’
```

The following example creates a view showing employees and the departments they belong to.

```
CREATE VIEW emp_dept
AS SELECT emp_lname, emp_fname, dept_name
FROM Employee JOIN Department
ON Employee.dept_id = Department.dept_id
```

CREATE WRITEFILE statement (deprecated)

Description

Use this statement to create a write file for a database.

Deprecated statement
The use of writefiles is deprecated.

Syntax

CREATE WRITEFILE *write-file-name*
FOR DATABASE *db-file-name* [**KEY** *key*]
[**LOG OFF** | **LOG ON** [*log-file-name* [**MIRROR** *mirror-file-name*]]]

write-file-name | *db-file-name* | *log-file-name* | *mirror-file-name* : *string*

Usage

Creates a database write file with the supplied name and attributes.

The file names (*write-file-name*, *db-file-name*, *log-file-name*, *mirror-file-name*) are strings containing operating system file names.

☞ For information on strings, see "Strings" on page 9.

If you specify no path, or a relative path, the file is created relative to the current working directory of the server.

You cannot create a write file for a database that is currently loaded.

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 server is running must have write permissions on the directories where files are created.

Not supported on Windows CE.

You must specify a KEY value if you want to create a writefile for a strongly encrypted database.

Side effects

An operating system file is created.

See also

- "CREATE DATABASE statement" on page 338
- "The Write File utility (deprecated)" [*ASA Database Administration Guide*, page 609]
- "Working with write files (deprecated)" [*ASA Database Administration Guide*, page 294]
- "DatabaseKey connection parameter [DBKEY]" [*ASA Database Administration Guide*, page 187]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement creates a write file.

```
CREATE WRITEFILE ’c:\\sybase\\my_db.wrt’
FOR DATABASE ’c:\\sybase\\my_db.db’
LOG ON ’e:\\logdrive\\my_db.log’
```

DEALLOCATE statement

Description	Use this statement to free resources associated with a cursor.
Syntax	**DEALLOCATE [CURSOR]** *cursor-name* *cursor-name* : *identifier*
Usage	Frees all memory associated with a cursor, including the data items, indicator variables, and the structure itself. This option has no effect in Adaptive Server Anywhere. It is provided for compatibility with Adaptive Server Enterprise and Microsoft SQL Server. In Adaptive Server Enterprise, the CURSOR keyword is required. In Microsoft SQL Server, the keyword is not permitted. Adaptive Server Anywhere recognizes both forms.
Permissions	None.
Side effects	None.
See also	♦ "DECLARE CURSOR statement [ESQL] [SP]" on page 436
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Supported by Adaptive Server Enterprise.

DEALLOCATE DESCRIPTOR statement [ESQL]

Description	Use this statement to free memory associated with a SQL descriptor area.
Syntax	**DEALLOCATE DESCRIPTOR** *descriptor-name* *descriptor-name* : *string*
Usage	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 264 ♦ "The SQL descriptor area (SQLDA)" [*ASA Programming Guide*, page 181] ♦ "SET DESCRIPTOR statement [ESQL]" on page 611
Standards and compatibility	♦ **SQL/92** Entry-level feature. ♦ **SQL/99** Core feature. ♦ **Sybase** Supported by Open Client/Open Server.
Example	For an example, see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 264.

Declaration section [ESQL]

Description

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;

Usage

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 will be used as host variables. Not all C declarations are valid inside a declaration section. See "Using host variables" [*ASA Programming Guide*, page 153] for more information.

Permissions

None.

See also

- "BEGIN statement" on page 312

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Compatible with Adaptive Server Enterprise.

Example

```
EXEC SQL BEGIN DECLARE SECTION;
char *emp_lname, initials[5];
int dept;
EXEC SQL END DECLARE SECTION;
```

DECLARE statement

Description

Use this statement to declare a SQL variable within a compound statement (BEGIN . . . END).

Syntax

DECLARE *variable-name data-type*

Usage

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 immediately following BEGIN. In a Transact-SQL procedure or trigger, there is no such restriction.

Standards and compatibility

- ♦ **SQL/92** Persistent Stored Module feature.
- ♦ **SQL/99** Persistent Stored Module feature.
- ♦ **Sybase** Supported by Adaptive Server Enterprise.
  - To be compatible with Adaptive Server Enterprise, the variable name must be preceded by an @.
  - In Adaptive Server Enterprise, a variable that is declared in a procedure or trigger exists for the duration of the procedure or trigger. In Adaptive Server Anywhere, if a variable 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).

Example

The following batch illustrates the use of the DECLARE statement and prints a message on the server window:

```
BEGIN
  DECLARE varname CHAR(61);
  SET varname = 'Test name';
  MESSAGE varname;
END
```

DECLARE CURSOR statement [ESQL] [SP]

Description

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** }]
| *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 flags 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 forwards 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, Adaptive Server Anywhere supplies the most efficient kind of cursor, which is an asensitive cursor.

☞ For more information, see "Asensitive cursors" [*ASA Programming Guide*, page 38].

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, Adaptive Server 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.

☞ For more information, see "Asensitive cursors" [*ASA Programming Guide*, page 38].

SCROLL A cursor declared SCROLL can use all formats of the FETCH statement. When a SCROLL cursor is requested, Adaptive Server Anywhere supplies a value-sensitive cursor.

☞ For more information, see "Value-sensitive cursors" [*ASA Programming Guide*, page 39].

Adaptive Server 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.

☞ For more information, see "Insensitive cursors" [*ASA Programming Guide*, page 35].

SENSITIVE A cursor declared SENSITIVE is sensitive to changes to membership or values of the result set.

For more information, see "Sensitive cursors" [*ASA Programming Guide*, page 36].

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 may not be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.

In response to any request for a cursor that specifies FOR UPDATE, Adaptive Server Anywhere provides either a value-sensitive cursor or an asensitive cursor. Insensitive and asensitive cursors are not updateable.

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 get_row_count(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.SYSTABLE ' ||
            'where table_id=' || string(id_value);
  begin
    declare crsr cursor using qry;

    open crsr;
    fetch crsr into tabname;
    close crsr;
  end
end
```

Usage

The DECLARE CURSOR statement declares a cursor with the specified name for a SELECT statement or a CALL statement.

Permissions

None.

Side effects

None.

See also

- "PREPARE statement [ESQL]" on page 563
- "OPEN statement [ESQL] [SP]" on page 553
- "EXPLAIN statement [ESQL]" on page 480
- "SELECT statement" on page 597
- "CALL statement" on page 318
- "FOR statement" on page 487

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

Example

The following example illustrates how to declare a scroll cursor in Embedded SQL:

```
EXEC SQL DECLARE cur_employee SCROLL CURSOR
FOR SELECT * FROM employee;
```

The following example illustrates how to declare a cursor for a prepared statement in Embedded SQL:

```
EXEC SQL PREPARE employee_statement
FROM 'SELECT emp_lname FROM employee';
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 emp_lname
      FROM employee;
  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]

Description

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*

Usage

Adaptive Server Anywhere supports a DECLARE CURSOR syntax that is not supported in Adaptive Server Enterprise. For information on the full DECLARE CURSOR syntax, see "DECLARE CURSOR statement [ESQL] [SP]" on page 436.

This section describes the overlap between the Adaptive Server Anywhere and Enterprise flavors of DECLARE CURSOR.

Permissions

None.

Side effects

None.

See also

♦ "DECLARE CURSOR statement [ESQL] [SP]" on page 436

Standards and compatibility

♦ **SQL/92** Entry-level feature. The FOR UPDATE and FOR READ ONLY options are Transact-SQL extensions.

♦ **SQL/92** Core feature. The FOR UPDATE and FOR READ ONLY options are Transact-SQL extensions.

♦ **Sybase** There are some features of the Adaptive Server Enterprise DECLARE CURSOR statement that are not supported in Adaptive Server Anywhere.

- Adaptive Server Enterprise supports cursors opened for update of a list of columns from the tables specified in the *select-statement*. This is not supported in Adaptive Server Anywhere.
- In the Watcom-SQL dialect, a DECLARE CURSOR statement in a procedure, trigger, or batch must immediately follow the BEGIN keyword. In the Transact-SQL dialect, there is no such restriction.
- In Adaptive Server Enterprise, when a cursor is declared in a procedure, trigger, or batch, it exists for the duration of the procedure, trigger, or batch. In Adaptive Server 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).

* CURSOR *type* (UNIQUE, NO SCROLL, and so on) and CURSOR FOR *statement-name* are not supported in Adaptive Server Anywhere.

DECLARE LOCAL TEMPORARY TABLE statement

Description

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

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 clause is useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

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.

PCTFREE 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, 200 bytes are reserved in each page.

Usage

The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table. For definitions of *column-definition*, *column-constraint*, and *table-constraint*, see “CREATE TABLE statement” on page 407.

Declared local temporary tables within compound statements exist within the compound statement. (See “Using compound statements” [*ASA SQL User’s Guide*, page 680]). Otherwise, the declared local temporary table exists until the end of the connection.

The rows of a declared temporary table are deleted when the table is explicitly dropped and when the table goes out of scope. You can also explicitly delete rows using TRUNCATE or DELETE.

Permissions

None.

Side effects

None.

See also

- ♦ "CREATE TABLE statement" on page 407
- ♦ "Using compound statements" [*ASA SQL User's Guide*, page 680]

Standards and compatibility

- ♦ **SQL/92** Conforms to the SQL/92 standard.
- ♦ **SQL/99** SQL/foundation feature outside of core SQL.
- ♦ **Sybase** Adaptive Server Enterprise does not support DECLARE TEMPORARY TABLE.

Example

The following example illustrates how to declare a temporary table in Embedded SQL:

```
EXEC SQL DECLARE LOCAL TEMPORARY TABLE MyTable (
  number INT
   );
```

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

Description

Use this statement to delete rows from the database.

Syntax

DELETE [**FIRST** | **TOP** *n*]
[**FROM**] [*owner*.]*table-name*
[**FROM** *table-list*]
[**WHERE** *search-condition*]

Usage

The DELETE statement deletes all the rows from the named table that satisfy the search condition. If no WHERE clause is specified, all rows from the named table are deleted.

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 operation.

The optional second FROM clause in the DELETE statement allows rows to be deleted based on joins. If the second FROM clause is present, the WHERE clause qualifies the rows of this second FROM clause. Rows are deleted from the table name given in the first 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 491.

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_1in 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.

Deleting a significant amount of data using the DELETE statement will also update column statistics.

Permissions

Must have DELETE permission on the table.

Side effects

None.

See also

- ♦ “TRUNCATE TABLE statement” on page 642
- ♦ “INSERT statement” on page 528
- ♦ “INPUT statement [Interactive SQL]” on page 523
- ♦ “FROM clause” on page 491

Standards and compatibility

- ♦ **SQL/92** Entry-level compliant. The use of more than one table in the FROM clause is a vendor extension.
- ♦ **SQL/99** Core feature. The use of more than one table in the FROM clause is a vendor extension.
- ♦ **Sybase** Supported by Adaptive Server Enterprise, including the vendor extension.

Example

Remove employee 105 from the database.

```
DELETE
FROM employee
WHERE emp_id = 105
```

Remove all data prior to 2000 from the **fin_data** table.

```
DELETE
FROM fin_data
WHERE year < 2000
```

Remove all orders from **sales_order_items** table if their ship date is older than 2001-01-01 and their region is Central.

```
DELETE
FROM sales_order_items
FROM sales_order
WHERE sales_order_items.id = sales_order.id
  and ship_date < '2001-01-01' and region ='Central'
```

DELETE (positioned) statement [ESQL] [SP]

Description

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*

Usage

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 updateable.

Permissions

Must have DELETE permission on tables used in the cursor.

Side effects

None.

See also

♦ "UPDATE statement" on page 650
♦ "UPDATE (positioned) statement [ESQL] [SP]" on page 655
♦ "INSERT statement" on page 528
♦ "PUT statement [ESQL]" on page 567

Standards and compatibility

♦ **SQL/92** Entry-level feature. The range of cursors that can be updated may contain vendor extensions if the ANSI_UPDATE_CONSTRAINTS option is set to OFF.

- **SQL/99** 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. Procedure and trigger use is supported only in Adaptive Server Anywhere.

Example

The following statement removes the current row from the database.

```
DELETE
WHERE CURRENT OF cur_employee
```

DESCRIBE statement [ESQL]

Description

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 will be 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 will be 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 **sqld** 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 will be derived from the select list item: if the item is a simple column name, it will be used; otherwise, a substring of the expression will be 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 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 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)" [*ASA Programming Guide*, page 181].

Usage

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 264).

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.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 264
- "DECLARE CURSOR statement [ESQL] [SP]" on page 436
- "OPEN statement [ESQL] [SP]" on page 553
- "PREPARE statement [ESQL]" on page 563

Standards and compatibility

- **SQL/92** Part of the SQL/92 standard. Some clauses are vendor extensions.
- **SQL/99** Core feature. Some clauses are vendor extensions.

- ♦ **Sybase** Some clauses supported by Open Client/Open Server.

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;
}
```

DISCONNECT statement [ESQL] [Interactive SQL]

Description	Use this statement to drop the current connection to a database.
Syntax	**DISCONNECT** [*connection-name* | **CURRENT** | **ALL**] *connection-name* : *identifier*, *string*, or *hostvar*
Usage	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. An implicit ROLLBACK is executed on connections that are dropped. ☞ For information on dropping connections other than the current connection, see "DROP CONNECTION statement" on page 457.
Permissions	None.
Side effects	None.
See also	♦ "CONNECT statement [ESQL] [Interactive SQL]" on page 332 ♦ "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 610
Standards and compatibility	♦ **SQL/92** Intermediate-level feature. ♦ **SQL/99** SQL/foundation feature outside of core SQL. ♦ **Sybase** Supported by Open Client/Open Server.

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

Description

Use this statement to remove objects from the database.

Syntax

DROP
{ **DATATYPE** | **DOMAIN** } *datatype-name*
| **DBSPACE** *dbspace-name*
| **EVENT** *event-name*
| **FUNCTION** [*owner*.]*function-name*
| **INDEX** [[*owner*.]*table-name*.]*index-name*
| **MESSAGE** *msgnum*
| **PROCEDURE** [*owner*.]*procedure-name*
| **TABLE** [*owner*.]*table-name*
| **TRIGGER** [[*owner*.]*table-name*.]*trigger-name*
| **VIEW** [*owner*.]*view-name*

Usage

The DROP statement removes the definition of the indicated database structure. If the structure is a dbspace, all tables in that dbspace must be dropped prior to dropping the dbspace. If the structure is a table, all data in the table is automatically deleted as part of the dropping process. Also, all indexes and keys for the table are dropped by the DROP TABLE statement.

DROP TABLE, DROP INDEX, and DROP DBSPACE are prevented whenever the statement affects a table that is currently being used by another connection.

DROP PROCEDURE and DROP FUNCTION are prevented when the procedure or function is in use by another connection.

DROP DATATYPE is prevented if the data type is used in a table. You must change data types on all columns defined on 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.

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.

Side effects

Automatic commit. Clears the Results tab in the Results pane in Interactive SQL. DROP TABLE and DROP INDEX close all cursors for the current connection.

DROP INDEX cannot be used to drop an index on a local temporary table. An attempt to do so will result in an Index not found error.

Local temporary tables is an exception; no commit is performed when one is dropped.

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 338
- "CREATE DOMAIN statement" on page 347
- "CREATE INDEX statement" on page 368
- "CREATE FUNCTION statement" on page 362
- "CREATE PROCEDURE statement" on page 373
- "CREATE TABLE statement" on page 407
- "CREATE TRIGGER statement" on page 419
- "CREATE VIEW statement" on page 428

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Adaptive Server Enterprise for those objects that exist in Adaptive Server Enterprise.

Example

- Drop the department table from the database.

```
DROP TABLE department
```

Drop the emp_dept view from the database.

```
DROP VIEW emp_dept
```

DROP DATABASE statement

Description	Use this statement to delete all database files associated with a database.
Syntax	**DROP DATABASE** *database-name* [**KEY** *key*]

Usage

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 a stored procedure.

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 in order 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 338
- "DatabaseKey connection parameter [DBKEY]" [*ASA Database Administration Guide*, page 187]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

Drop the database *temp.db*, in the *C:\temp* directory..

```
DROP DATABASE 'c:\temp\temp.db'
```

DROP CONNECTION statement

Description	Use this statement to drop a connection to the database, belonging to any user.
Syntax	**DROP CONNECTION** *connection-id*

Usage

The DROP CONNECTION statement disconnects a user from the database by dropping the connection to the database.

You can obtain the *connection-id* by using the **connection_property** function to request the connection number. The following statement returns the connection ID of the current connection:

```
SELECT connection_property( 'number' )
```

Permissions

Must have DBA authority.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 332

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement drops the connection with ID number 4.

```
DROP CONNECTION 4
```

DROP EXTERNLOGIN statement

Description

Use this statement to drop an external login from the Adaptive Server 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.

Usage

DROP EXTERNLOGIN deletes an external login from the Adaptive Server Anywhere catalogs.

Permissions

Must be the owner of *login-name* or have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE EXTERNLOGIN statement" on page 360

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

```
DROP EXTERNLOGIN DBA TO sybase1
```

DROP PUBLICATION statement

Description

Use this statement to drop a publication. In MobiLink a publication identifies synchronized data in a Adaptive Server 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*

Usage

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" on page 280
- "CREATE PUBLICATION statement" on page 385
- "sp_drop_publication procedure" [*SQL Remote User's Guide,* page 387]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Example

The following statement drops the pub_contact publication.

```
DROP PUBLICATION pub_contact
```

DROP REMOTE MESSAGE TYPE statement [SQL Remote]

Description	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**
Usage	The statement removes a message type from a database.
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 388 ♦ "sp_drop_remote_type procedure" [*SQL Remote User's Guide*, page 388] ♦ "Using message types" [*SQL Remote User's Guide*, page 210].
Example	The following statement drops the FILE message type from a database.

```
DROP REMOTE MESSAGE TYPE file
```

DROP SERVER statement

Description

Use this statement to drop a remote server from the Adaptive Server Anywhere catalog.

Syntax

DROP SERVER *server-name*

Usage

DROP SERVER deletes a remote server from the Adaptive Server Anywhere catalogs. You must drop all the proxy tables that have been defined for the remote server before this statement will succeed.

Permissions

Only the DBA account can delete a remote server.

Not supported on Windows CE.

Side effects

Automatic commit.

See also

- "CREATE SERVER statement" on page 392

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

```
DROP SERVER ase_prod
```

DROP SERVICE statement

Description	Use this statement to permit a database server to act as a web server.
Syntax	**DROP SERVICE** *service-name*
Usage	This statement deletes a web service.
Permissions	Must have DBA authority.
Side effects	None.
See also	♦ “ALTER SERVICE statement” on page 286, “CREATE SERVICE statement” on page 395
Example	To drop a web service named tables, execute the following statement:

```
CREATE SERVICE tables
```

DROP STATEMENT statement [ESQL]

Description

Use this statement to free statement resources.

Syntax

DROP STATEMENT [*owner*.]*statement-name*

statement-name : *identifier* | *hostvar*

Usage

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 563

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Open Client/Open Server

Example

The following are examples of DROP STATEMENT use:

```
EXEC SQL DROP STATEMENT S1;
EXEC SQL DROP STATEMENT :stmt;
```

DROP STATISTICS statement

Description	Use this statement to erase all optimizer statistics on the specified columns.
Syntax	**DROP STATISTICS** [**ON**] [*owner.*]*table-name* [**(** *column-list* **)**]

Usage

The Adaptive Server Anywhere optimizer uses statistical information to determine the best strategy for executing each statement. Adaptive Server Anywhere automatically gathers and updates these statistics. These statistics are stored permanently in the database in the system table SYSCOLSTAT. 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.

The DROP STATISTICS statement deletes all internal statistical data from the system table SYSCOLSTAT for the specified columns. This drastic step leaves the optimizer with no access to essential statistical information. Without these statistics, the optimizer may generate very inefficient data access plans, causing poor database performance.

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 399

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

DROP SUBSCRIPTION statement [SQL Remote]

Description

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 may 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 may have more than one subscription to a publication.

subscriber-id The user ID of the subscriber to the publication.

Usage

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.

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Adaptive Server Anywhere version 7.0.

See also

- ♦ "CREATE SUBSCRIPTION statement [SQL Remote]" on page 400

Example

The following statement drops a subscription for the user ID **SamS** to the publication **pub_contact**.

```
DROP SUBSCRIPTION TO pub_contact
FOR SamS
```

DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Description

Use this statement to drop a synchronization subscription within a MobiLink remote database or a MobiLink reference database. You can also use it to drop a default subscription, which contains default subscription values, for the specified publication.

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 subscription for the publication. MobiLink users subscribed to a publication inherit as defaults the values in a default publication.

Usage

Drop a synchronization subscription in a MobiLink remote or reference database.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side Effects

Automatic commit.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Examples

Unsubscribe MobiLink user ml_user1 to the sales publication.

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR "ml_user1"
```

Drop the default subscription, which contains default subscription values, for the sales publication (by omitting the FOR clause).

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
```

DROP SYNCHRONIZATION USER statement [MobiLink]

Description	Use this statement to drop a synchronization user from a MobiLink remote database.
Syntax	**DROP SYNCHRONIZATION USER** *ml_username*, . . . *ml_username*: *identifier*
Usage	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.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension.
Example	Remove MobiLink user ml_user1 from the database.

```
DROP SYNCHRONIZATION USER ml_user1
```

DROP VARIABLE statement

Description	Use this statement to eliminate a SQL variable.
Syntax	**DROP VARIABLE** *identifier*
Usage	The DROP VARIABLE statement eliminates a SQL variable that was previously created using the CREATE VARIABLE statement. Variables will be 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 may free up significant resources (primarily disk space).
Permissions	None.
Side effects	None.
See also	♦ "CREATE VARIABLE statement" on page 427 ♦ "SET statement" on page 604
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not supported in Adaptive Server Enterprise.

EXCEPT operation

Description

Computes the difference between the result sets of two or more queries.

Syntax

select-statement
  EXCEPT [**ALL** | **DISTINCT**] *select-statement*
[**EXCEPT** [**ALL** | **DISTINCT**] *select-statement*] . . .
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], . . .]

Usage

The differences between the result sets of several SELECT statements can be obtained as a single result using EXCEPT or EXCEPT ALL. EXCEPT DISTINCT is identical to EXCEPT.

The component SELECT statements must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

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, Adaptive Server Anywhere will choose a data type for the corresponding column in the result and automatically convert the columns in each component SELECT statement 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 SELECT statement. An alternative way of customizing result set column names is to use the WITH clause on the SELECT statement.

Permissions

Must have SELECT permission for each of the component SELECT statements.

Side effects

None

See also

"INTERSECT operation" on page 534

"UNION operation" on page 644

Standards and compatibility

- **SQL/92** Entry-level.
- **SQL/99** EXCEPT DISTINCT is a core feature. EXCEPT ALL is feature F304.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

For examples of EXCEPT usage, see "Set operators and NULL" [*ASA SQL User's Guide,* page 256].

EXECUTE statement [ESQL]

Description

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 may 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.

Usage

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" [*ASA Programming Guide*, page 167]).

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 an 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 475
- "PREPARE statement [ESQL]" on page 563
- "DECLARE CURSOR statement [ESQL] [SP]" on page 436

Standards and compatibility

- **SQL/92** Intermediate-level feature.
- **SQL/99** Feature outside of core SQL.
- **Sybase** Supported in Open Client/Open Server.

Example

Execute a DELETE.

```
EXEC SQL EXECUTE IMMEDIATE
'DELETE FROM employee WHERE emp_id = 105';
```

Execute a prepared DELETE statement.

```
EXEC SQL PREPARE del_stmt FROM
'DELETE FROM employee WHERE emp_id = :a';
EXEC SQL EXECUTE del_stmt USING :employee_number;
```

Execute a prepared query.

```
EXEC SQL PREPARE sel1 FROM
'SELECT emp_lname FROM employee WHERE emp_id = :a';
EXEC SQL EXECUTE sel1 USING :employee_number INTO :emp_lname;
```

EXECUTE statement [T-SQL]

Description

Use Syntax 1 to invoke a procedure, as an Adaptive Server Enterprise-compatible alternative to the CALL statement. You can also execute statements *within* Transact-SQL stored procedures and triggers. For more information, see "EXECUTE IMMEDIATE statement [SP]" on page 475. 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* **)**

Usage

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.

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 318
- "EXECUTE statement [ESQL]" on page 471
- "EXECUTE IMMEDIATE statement [SP]" on page 475

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 database server 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]

Description

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 QUOTED_IDENTIFIER is set to OFF.

☞ For more information, see the "QUOTED_IDENTIFIER option [compatibility]" [*ASA Database Administration Guide*, page 685].

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 filenames 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.

Usage

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" [*ASA SQL User's Guide*, page 704].

See also

- ♦ "CREATE PROCEDURE statement" on page 373
- ♦ "BEGIN statement" on page 312
- ♦ "EXECUTE statement [ESQL]" on page 471

Standards and compatibility

- ♦ **SQL/92** Intermediate-level feature.

- ♦ **SQL/99** SQL/foundation feature outside of core SQL.
- ♦ **Sybase** Supported in Open Client/Open Server.

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" [*ASA SQL User's Guide*, page 704].

EXIT statement [Interactive SQL]

Description

Use this statement to leave Interactive SQL.

Syntax

{ **EXIT** | **QUIT** | **BYE** } [*return-code*]

return-code: *number* | *connection-variable*

Usage

This statement closes your connection with the database, then closes the Interactive SQL environment. 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 613

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable in Adaptive Server Enterprise.

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

```
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;
```

because EXIT is an Interactive SQL statement (not a SQL statement), and you cannot include any Interactive SQL statement in other SQL block statements.

The following Windows batch file prints `Error = 1` on the command prompt.

```
dbisql -c "dsn=ASA 9.0 Sample" EXIT 1
if errorlevel 1 echo "Errorlevel is 1"
```

EXPLAIN statement [ESQL]

Description

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*

Usage

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 will be accessed by the database server. After each table is a parenthesized index name. This is the index that will be used to access the table. If no index will be used (the table will be 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[3]** field of the SQLCA (SQLIOESTIMATE) will be 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” [*ASA SQL User’s Guide*, page 157].

Permissions

Must have opened the named cursor.

Side effects

None.

See also

- “DECLARE CURSOR statement [ESQL] [SP]” on page 436
- “PREPARE statement [ESQL]” on page 563

- "FETCH statement [ESQL] [SP]" on page 482
- "CLOSE statement [ESQL] [SP]" on page 325
- "OPEN statement [ESQL] [SP]" on page 553
- "Using cursors in embedded SQL" [*ASA Programming Guide*, page 167]
- "The SQL Communication Area (SQLCA)" [*ASA Programming Guide*, page 161]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.

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 emp_id, emp_lname
   FROM employee
   WHERE emp_lname 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:

```
'employee <seq>'
```

FETCH statement [ESQL] [SP]

Description

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 backwards. 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" [*ASA SQL User's Guide*, page 692]).

  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 of work tables in query processing" [*ASA SQL User's Guide*, page 190] 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 server. The client buffers these rows, and subsequent fetches are retrieved from these buffers without a new request to the 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 will be sent to the server and a single row will be 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. For more information, see "OPEN statement [ESQL] [SP]" on page 553.

FETCH RELATIVE 0 always re-fetches the row.

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 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 a write lock on the row. The lock will be held until the end of the current transaction. See "How locking works" [*ASA SQL User's Guide*, page 135].

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 the section "Fetching more than one row at a time" [*ASA Programming Guide*, page 170].

Usage

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.

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 436
- "PREPARE statement [ESQL]" on page 563
- "OPEN statement [ESQL] [SP]" on page 553
- "Using cursors in embedded SQL" [*ASA Programming Guide*, page 167]
- "Using cursors in procedures and triggers" [*ASA SQL User's Guide*, page 692]
- FETCH in *PowerScript Reference*

Standards and compatibility

- **SQL/92** Entry-level feature. Use in procedures is a Persistent Stored Module feature.
- **SQL/99** Core feature. Use in procedures is a Persistent Stored Module feature.
- **Sybase** Supported in Adaptive Server Enterprise.

Example

The following is an Embedded SQL example.

```
EXEC SQL DECLARE cur_employee CURSOR FOR
SELECT emp_id, emp_lname FROM employee;
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 emp_lname
      FROM employee;
   DECLARE name CHAR(40);
   OPEN cur_employee;
   LOOP
      FETCH NEXT cur_employee into name;
       ...
   END LOOP
   CLOSE cur_employee;
END
```

FOR statement

Description

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 FOR *statement*
  [**FOR UPDATE** | **FOR READ ONLY**]
    DO *statement-list*
END FOR [*statement-label*]

Usage

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 will be the data types of the expressions in the select list. The names will be the select list item aliases, if they exist; otherwise, they will be 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 will be 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 436
- "FETCH statement [ESQL] [SP]" on page 482
- "LEAVE statement" on page 535
- "LOOP statement" on page 548

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following fragment illustrates the use of the FOR loop.

```
FOR names AS curs CURSOR FOR
SELECT emp_lname
FROM employee
DO
   CALL search_for_name( emp_lname );
END FOR;
```

FORWARD TO statement

Description

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*]

Usage

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 Adaptive Server 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 server uses:

- A remote login alias set using CREATE EXTERNLOGIN, or
- If a remote login alias is not set up, the name and password used to communicate with Adaptive Server 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 ({}).

Note
The FORWARD TO statement is a server directive and cannot be used within procedures 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.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Open Client/Open Server.

Example

The following example shows a passthrough session with the remote server ase_prod:

```
FORWARD TO aseprod
  'SELECT * FROM titles;  SELECT * FROM authors';
FORWARD TO
```

FROM clause

Description

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
| *view*
| *procedure*
| *derived-table*
| *lateral-derived-table*
| *joined table*
| **(** *table-expression*, . . . **)**

table :
[*userid* **.**] *table-name*
[[**AS**] *correlation-name*]
[**WITH (** *table-hint* | **INDEX (** *index-name* **))** | **FORCE INDEX (** *index-name* **)**]

view :
[*userid* **.**] *view-name* [[**AS**] *correlation-name*]
[**WITH (** *table-hint* **)**]

procedure :
[*owner*.]*procedure-name* **(** [*parameter*, . . .] **)**
[**WITH(** *column-name data-type*, ... **)**]
[[**AS**] *correlation-name*]

derived-table :
(*select-statement* **)**
[**AS**] *correlation-name* [**(** *column-name*, . . . **)**]

lateral-derived-table :
LATERAL (*select-statement* | *table-expression* **)**
[**AS**] *correlation-name* [**(** *column-name*, . . . **)**]

joined table:
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:
  NOLOCK
| XLOCK
| READUNCOMMITTED
| READCOMMITTED
| REPEATABLEREAD
| HOLDLOCK
| SERIALIZABLE
| FASTFIRSTROW
```

Parameters

table 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 will be found by default without specifying the user ID (see "Referring to tables owned by groups" [*ASA Database Administration Guide*, page 447]).

The **WITH (INDEX (** *index-name* **))** or the equivalent **FORCE INDEX (** *index-name* **)** clauses 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.

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 hence to poor performance.

view 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 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.customer, sp.quantity, product.name
FROM sp_customer_products( 149 ) WITH ( customer int, quantity
         int ) sp
   JOIN product
ON sp.customer = product.id
```

derived-table 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 A derived table, stored procedure, or joined table that may include outer references. You must use a lateral derived table if you wish to use an outer reference in the FROM clause. For information about outer references, see "Outer references" [*ASA SQL User's Guide*, page 368].

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
```

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, in:

```
SELECT *
FROM sales_order
KEY JOIN sales_order_items,
sales_order
KEY JOIN employee
```

the two instances of the **sales_order** table are treated as one instance, and so is equivalent to:

```
SELECT *
FROM sales_order
KEY JOIN sales_order_items
KEY JOIN employee
```

Whereas:

```
SELECT *
FROM Person HUSBAND, Person WIFE
```

would be treated as two instances of the Person table, with different correlation names HUSBAND and WIFE.

WITH table-hint allows you to specify the behavior of Adaptive Server Anywhere to be used only for this table, and only for this statement. You can use WITH *table-hint* to change Adaptive Server Anywhere's 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
WITH table-hint 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 hints** The following table hints can be used to specify isolation level settings for tables. They specify a locking method to be used only for this table, and only for this statement.

  The table hints set the following isolation levels:

Table hint	Isolation level
NOLOCK	0
READUNCOMMITTED	0
READCOMMITTED	1
REPEATABLEREAD	2
HOLDLOCK	3
SERIALIZABLE	3
XLOCK	All isolation levels

Note
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.

- **Optimization hints** 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, Adaptive Server Anywhere chooses an access plan that is intended to reduce the time to fetch the first row of the query's result.

☞ For more information, see "OPTIMIZATION_GOAL option [database]" [*ASA Database Administration Guide*, page 676].

Usage

The SELECT, UPDATE, and DELETE statements require a table list, to specify which tables are used by the statement.

Views and derived tables
Although this 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 445
- "SELECT statement" on page 597
- "UPDATE statement" on page 650
- "Joins: Retrieving Data from Several Tables" [*ASA SQL User's Guide*, page 263]

Standards and compatibility

- **SQL/92** Entry-level feature. The complexity of the FROM clause means that you should check individual clauses against the standard.

- **SQL/99** Core feature, except for KEY JOIN, which is a vendor extension; and FULL OUTER JOIN and NATURAL JOIN, which are SQL/foundation features outside of core SQL. The complexity of the FROM clause means that you should check individual clauses against the standard.

- **Sybase** The ON phrase is not supported in Adaptive Server Enterprise prior to version 12. In earlier versions, you must use the WHERE clause to build joins.

Example

The following are valid FROM clauses:

```
...
FROM employee
...
...
FROM employee NATURAL JOIN department
...
```

```
...
FROM customer
KEY JOIN sales_order
KEY JOIN sales_order_items
KEY JOIN product
...
```

The following query illustrates how to use derived tables in a query:

```
SELECT lname, fname, number_of_orders
FROM customer JOIN
     ( SELECT cust_id, count(*)
       FROM sales_order
        GROUP BY cust_id )
     AS sales_order_counts ( cust_id,
                             number_of_orders )
ON ( customer.id = sales_order_counts.cust_id )
WHERE number_of_orders > 3
```

The following query illustrates how to select rows from stored procedure result sets:

```
SELECT t.id, t.quantity_ordered AS q, p.name
FROM sp_customer_products( 149 ) t JOIN product p
ON t.id = p.id
```

GET DATA statement [ESQL]

Description

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 604.

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 LONGVARCHAR or a LONGVARCHAR 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.

Usage

Get a piece of one column value from the row at the current cursor position.

Permissions

The cursor must be opened and positioned on a row, using FETCH.

Side effects

None.

See also

- "FETCH statement [ESQL] [SP]" on page 482
- "READTEXT statement [T-SQL]" on page 573

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Open Client/Open Server. An alternative is the Transact-SQL READTEXT statement.

Example

The following example uses GET DATA to fetch a **binary large object** (often 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]

Description

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**

Usage

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 will be retrieved. Type checking is performed when doing GET . . . DATA to ensure that the host variable and the descriptor variable have the same data type. LONGVARCHAR and LONGBINARY 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 264
- "DEALLOCATE DESCRIPTOR statement [ESQL]" on page 433
- "SET DESCRIPTOR statement [ESQL]" on page 611
- "The SQL descriptor area (SQLDA)" [*ASA Programming Guide*, page 181]

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

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 264.

GET OPTION statement [ESQL]

Description

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*

Usage

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 will be 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 613
- "System and catalog stored procedures" on page 777
- "CONNECTION_PROPERTY function [System]" on page 117

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement illustrates use of GET OPTION.

```
EXEC SQL GET OPTION 'date_format' INTO :datefmt;
```

GOTO statement [T-SQL]

Description

Use this statement to branch to a labeled statement.

Syntax

label : **GOTO** *label*

Usage

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/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Adaptive Server Enterprise supports the GOTO statement.

Example

The following Transact-SQL batch prints the message “yes” on the server window four times:

```
declare @count smallint
select @count = 1
restart:
   print ’yes’
   select @count = @count + 1
   while @count <=4
    goto restart
```

GRANT statement

Description

Use this statement to create new user IDs, to grant or deny permissions to specific users, and to create or change passwords.

Syntax 1

GRANT CONNECT TO *userid*, . . .
[**AT** *starting-id*]
IDENTIFIED BY *password*, . . .

Syntax 2

GRANT {
 DBA,
 GROUP,
 MEMBERSHIP IN GROUP *userid*, . . . ,
 [**RESOURCE** | **ALL**]
}
  TO *userid*, . . .

Syntax 3

GRANT {
 ALL [**PRIVILEGES**],
 ALTER,
 DELETE,
 INSERT,
 REFERENCES [**(** *column-name*, . . . **)**],
 SELECT [**(** *column-name*, . . . **)**],
 UPDATE [**(** *column-name*, . . . **)**],
}
 ON [*owner*.]*table-name*
 TO *userid*, . . .
  [**WITH GRANT OPTION**]
  [**FROM** *userid*]

Syntax 4

GRANT EXECUTE ON [*owner*.]*procedure-name* **TO** *userid*, . . .

Syntax 5

GRANT INTEGRATED LOGIN TO *user_profile_name*, . . . **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, type:

```
GRANT CONNECT TO userid IDENTIFIED BY ""
```

To create a user with no password, type:

```
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. The password must be a valid identifier, as described in

"Identifiers" on page 7. Passwords have a maximum length of 255 bytes.

The following are invalid for database user IDs and passwords:

- ♦ names that begin with white space or single or double quotes
- ♦ names that end with white space
- ♦ names that contain semicolons

It is recommended that the password be composed of 7-bit ASCII characters as 8-bit ASCII characters above 127 may not work correctly when character set translation is turned on and the character sets do not match.

If you want to use characters above 127, you should do one of the following:

- ♦ specify **CharSet=none** in your connection string to turn off character set translation
- ♦ specify the character set using the CharSet connection parameter
- ♦ specify the -ct- option to turn off character set translation on the database server

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.

GROUP Allows the user(s) to have members.

☞ For more information, see "Managing groups" [*ASA Database Administration Guide,* page 444].

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.

☞ For more information, see "Managing groups" [*ASA Database Administration Guide,* page 444].

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 all of the permissions outlined below.

ALTER The users will be allowed to alter the named table with the ALTER TABLE statement. This permission is not allowed for views.

DELETE The users will be allowed to delete rows from the named table or view.

INSERT The users will be allowed to insert rows into the named table or view.

REFERENCES [(column-name, . . .)] The users will be allowed to create indexes on the named table, and foreign keys which reference the named tables. If column names are specified, the users will be 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 will be allowed to look at information in this view or table. If column names are specified, the users will be allowed to look at only those columns. SELECT permissions on columns cannot be granted for views, only for tables.

UPDATE [(column-name, . . .)] The users will be allowed to update rows in this view or table. If column names are specified, the users will be allowed to update only those columns. UPDATE permissions on columns cannot be granted for views, only for tables.

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.

Usage

The GRANT statement is used to grant database permissions to individual user IDs and groups. It is also used to create and delete 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.

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 machine to connect to a database without having to provide a user ID or password.

☞ For more information on integrated logins, see "Using integrated logins" [*ASA Database Administration Guide,* page 74].

Permissions

Syntax 1 or 2 One of the following conditions must be met.

- You are changing your own password using GRANT CONNECT.
- You 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, at least one of the following conditions must be met:

- You own the table
- You have been granted permissions on the table with GRANT OPTION
- You have DBA authority

Syntax 4 One of the following conditions must be met:

- You own the procedure
- You have DBA authority

Syntax 5 The following condition must be met:

- You have DBA authority

Side effects

Automatic commit.

See also

- "REVOKE statement" on page 585

Standards and compatibility

- **SQL/92** Syntax 3 is an entry-level feature. Syntax 4 is a Persistent Stored Module feature. Other syntaxes are vendor extensions.
- **SQL/99** Syntax 3 is a core feature. Syntax 4 is a Persistent Stored Module feature. Other syntaxes are vendor extensions.
- **Sybase** Syntaxes 2 and 3 are supported in Adaptive Server Enterprise. The security model is different in Adaptive Server Enterprise and Adaptive Server Anywhere, so other syntaxes differ.

Example

Make two new users for the database.

```
GRANT
CONNECT TO Laurel, Hardy
IDENTIFIED BY Stan, Ollie
```

Grant permissions on the employee table to user Laurel.

```
GRANT
SELECT, UPDATE ( street )
ON employee
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]

Description

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.

Usage

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. In order 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 will be run periodically, and that the consolidated database will have no SEND clause specified.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 510
- "GRANT REMOTE statement [SQL Remote]" on page 511
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 587
- "sp_grant_consolidate procedure" [*SQL Remote User's Guide*, page 390]

Example

```
GRANT CONSOLIDATE TO con_db
TYPE mapi
ADDRESS 'Consolidated Database'
```

GRANT PUBLISH statement [SQL Remote]

Description

Use this statement to identify the publisher of the current database.

Syntax

GRANT PUBLISH TO *userid*

Usage

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.

In order 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 510
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 508
- "REVOKE PUBLISH statement [SQL Remote]" on page 588
- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 400
- "sp_publisher procedure" [*SQL Remote User's Guide*, page 409]

Example

```
GRANT PUBLISH TO publisher_ID
```

GRANT REMOTE statement [SQL Remote]

Description

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*.

Usage

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. In order 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 will be 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.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- ♦ "GRANT PUBLISH statement [SQL Remote]" on page 510
- ♦ "REVOKE REMOTE statement [SQL Remote]" on page 589
- ♦ "GRANT CONSOLIDATE statement [SQL Remote]" on page 508
- ♦ "sp_grant_remote procedure" [*SQL Remote User's Guide,* page 393]
- ♦ "Granting and revoking REMOTE and CONSOLIDATE permissions" [*SQL Remote User's Guide,* page 204]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.

Example

- ♦ The following statement grants remote permissions to user **SamS**, using a MAPI e-mail 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 [SQL Remote]

Description

Use this statement to provide DBA privileges to a user ID, but only when connected from the Message Agent.

Syntax

GRANT REMOTE DBA
TO *userid*, . . .
IDENTIFIED BY *password*

Usage

REMOTE DBA authority enables the Message Agent to have full access to the database in order to make any changes contained in the messages, while avoiding security problems associated with distributing DBA user IDs passwords.

REMOTE DBA 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

- "The Message Agent and replication security" [*SQL Remote User's Guide*, page 243]
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 590

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

GROUP BY clause

Description

Use this clause to group columns, alias names, and functions as a part of the SELECT statement.

Syntax

GROUP BY [*group-by-list*
| **ROLLUP (** *group-by-list* **)**
| **(** *group-by-list* **) WITH ROLLUP**
| **CUBE (** *group-by-list* **)**
| **(** *group-by-list* **) WITH CUBE**
| **GROUPING SETS (** *grouping-sets* **)**]
[**HAVING** *search-condition*]

Parameters

ROLLUP keyword The ROLLUP operation adds summary (or subtotalled) rows into the result set of a query with a GROUP BY clause.

A **prefix** is a subset of the items in the *group-by-list.* A prefix is defined by excluding one or more of the rightmost items from those in the *group-by-list.* An additional row is added to the result set for each prefix. The additional row contains subtotal information for a set of rows in the GROUP BY result set. Each subtotal row includes the following:

- **Column excluded from the prefix** NULL.
- **Column included in the prefix** The value of the column.
- **Aggregate function** An aggregate over the values of the excluded columns.

For more information about ROLLUP operations, see "Using ROLLUP" [*ASA SQL User's Guide,* page 335].

WITH ROLLUP keyword This is an alternate way to construct ROLLUP queries. It returns the same result set as the ROLLUP syntax. For more information about ROLLUP operations, see "Using ROLLUP" [*ASA SQL User's Guide,* page 335].

CUBE keyword The CUBE operation generates subtotal rows for a GROUP BY by considering grouping across all possible subsets of the group-by-list. For more information about CUBE operations, see "Using CUBE" [*ASA SQL User's Guide,* page 340].

WITH CUBE keyword This is an alternate way to construct CUBE queries. It returns the same result set as the CUBE syntax. For more information about CUBE operations, see "Using CUBE" [*ASA SQL User's Guide,* page 340].

GROUPING SETS keyword The GROUPING SETS operation lets you compute groups on several different sets of grouping columns (called

GROUPING SETS) in the same query. The items in parentheses each represent a family of one or more GROUPING SETS. All GROUPING SETS contained in these families are computed when the GROUPING SETS keyword is used. Rows returned for GROUPING SETS that do not group by all non-aggregate columns of the *select-list* will contain NULL values in the non-grouping, non-aggregate columns. To distinguish between NULL values present in the input data and NULL values inserted by the grouping operator, use the GROUPING() aggregate function. For more information about the GROUPING SETS operation, see "Using GROUPING SETS" [*ASA SQL User's Guide*, page 343].

Note
In a GROUPING SETS query, **( )** denotes a GROUPING SET that uses no grouping columns and therefore generates a single row of output in the result set.

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.

Usage

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. All NULL-containing rows are treated as a single set. The resulting rows are often referred to as groups since there is one row in the result for each group of rows from the table list. Aggregate functions can then be applied to these groups to get meaningful results.

Note
When GROUP BY is used, 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.

See also

- ♦ "Expressions" on page 16
- ♦ "SELECT statement" on page 597
- ♦ "FROM clause" on page 491
- ♦ "Search conditions" on page 23
- ♦ "UNION operation" on page 644
- ♦ "Joins: Retrieving Data from Several Tables" [*ASA SQL User's Guide*, page 263]
- ♦ "About OLAP operations" [*ASA SQL User's Guide*, page 331]
- ♦ "Using ROLLUP" [*ASA SQL User's Guide*, page 335]

- "Using CUBE" [*ASA SQL User's Guide*, page 340]
- "Using GROUPING SETS" [*ASA SQL User's Guide*, page 343]

Standards and compatibility

- **SQL/92** Entry-level feature. The complexity of the SELECT statement means that you should check individual clauses against the standard.
- **SQL/99** 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.
- **Sybase** Supported by Adaptive Server Enterprise, with some differences in syntax. OLAP features are not supported in Adaptive Server Enterprise.

Examples

The following ROLLUP query 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 (order_date) Year, quarter (order_date) Quarter,
         count (*) Orders
FROM sales_order
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 (order_date) Year, quarter (order_date) Quarter,
         count (*) Orders
FROM sales_order
GROUP BY CUBE (Year, Quarter)
ORDER BY Year, Quarter
```

The following GROUPING SETS query 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 subtotalled instead of returning all combinations of subtotals like the CUBE operation.

```
SELECT year (order_date) Year, quarter (order_date) Quarter,
         count (*) Orders
 FROM sales_order
GROUP BY GROUPING SETS ((Year, Quarter), (Year))
ORDER BY Year, Quarter
```

HELP statement [Interactive SQL]

Description	Use this statement to receive help in the Interactive SQL environment.
Syntax	**HELP** [*topic*]
Usage	The HELP statement is used to access SQL Anywhere Studio documentation. The *topic* for help can be optionally specified. If *topic* is a reserved word, it must be enclosed in single quotes. In some help formats, the topic cannot be specified; in this case, a link to the home page of the online books is provided.
Permissions	None.
Side effects	None.
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not applicable

IF statement

Description

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

Usage

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 18.

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 312
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** The Transact-SQL IF statement has a slightly different syntax.

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 company_name, CAST(    sum(sales_order_items.quantity
         *
   product.unit_price) AS INTEGER) VALUE
   FROM customer
   LEFT OUTER JOIN sales_order
   LEFT OUTER JOIN sales_order_items
   LEFT OUTER JOIN product
   GROUP BY company_name;
   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]

Description

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]

Usage

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/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Adaptive Server Enterprise supports the Transact-SQL IF statement.

Example

The following example illustrates the use of the Transact-SQL IF statement:

```
IF (SELECT max(id) FROM sysobjects) < 100
   RETURN
ELSE

      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]

Description	Use this statement to include a file into a source program to be scanned by the SQL preprocessor.
Syntax	**INCLUDE** *filename* *filename* : **SQLDA** | **SQLCA** | *string*

Usage

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/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

INPUT statement [Interactive SQL]

Description

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** *filename* | **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

FROM clause The *filename* is passed to the server as a quoted string. The string is therefore 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 employee table is:

  ```
  INPUT INTO employee
  FROM 'c:\\temp\\input.dat'
  ```

- The path name is relative to the machine 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 appears where the user can enter the values for the new row. If the user is running Interactive SQL on the command line, then Interactive SQL prompts the user 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. When input is entered by the user, a dialog is provided for the user to enter one row per line in the input format.

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, 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 ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas 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. Optionally, 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.

- **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 doesn't exist, it will be created.

- **DBASEII** The file is in dBASE II format. If the table doesn't exist, it will be created.

- **DBASEIII** The file is in dBASE III format. If the table doesn't exist, it will be created.

- **EXCEL** Input file is in the format of Microsoft Excel 2.1. If the table doesn't exist, it will be 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 doesn't exist, it will be 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 doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct because the information in the file pertains to a cell, not to a column.

ESCAPE CHARACTER clause The default escape character for hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example.

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 '!'
```

Only one single-byte character can be used as an escape character.

ESCAPES clause 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.

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 lists (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, 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.

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 will be 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.

If the ASCII file has entries such that a column appears to be null, it is treated as NULL. If the column in that position cannot be NULL, a zero is inserted in numeric columns and an empty string in character columns.

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.

If *encoding* is not specified, Interactive SQL determines the code page that is used to read the file as follows, where code page values occurring earlier in the list take precedence over those occurring later in the list:

- ♦ the code page specified with the DEFAULT_ISQL_ENCODING option (if this option is set)
- ♦ the code page specified with the -codepage option when Interactive SQL was started
- ♦ the default code page for the computer Interactive SQL is running on

☞ For more information about Interactive SQL and encodings, see "DEFAULT_ISQL_ENCODING option [Interactive SQL]" [*ASA Database Administration Guide,* page 649].

Usage

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 filename is specified). If neither is specified, the input will be read from the command file that contains the INPUT statement—in Interactive SQL, this can even be directly from the SQL Statements pane. In this case, input is ended with a line containing only the string END.

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 in the Messages pane 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.

Permissions

Must have INSERT permission on the table or view.

Side effects

None.

See also

- "OUTPUT statement [Interactive SQL]" on page 556
- "INSERT statement" on page 528
- "UPDATE statement" on page 650
- "DELETE statement" on page 445
- "SET OPTION statement" on page 613
- "LOAD TABLE statement" on page 538

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

The following is an example of an INPUT statement from an ASCII text file.

```
INPUT INTO employee
FROM new_emp.inp
FORMAT ASCII;
```

INSERT statement

Description

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** }]
VALUES (*expression* | **DEFAULT**, . . . **)**

Syntax 2

INSERT [**INTO**] [*owner*.]*table-name* [**(** *column-name*, . . . **)**]
[**ON EXISTING** { **ERROR** | **SKIP** | **UPDATE** }]
[**WITH AUTO NAME**]
select-statement

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 will be 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 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 server does a primary key lookup for each input row. If the corresponding row does not already exist in the table, it inserts the new row as usual. For rows that already exist in the table, you can choose to silently ignore the input row (SKIP), update the values in the input row (UPDATE), or generate an error message for duplicate key values (ERROR).

By default, if you do not specify ON EXISTING, attempting to insert rows into a table where the row already exist results in a duplicate key value error. This is equivalent to specifying ON EXISTING ERROR.

Usage

The INSERT statement is used to add new rows to a database table.

Syntax 1 Insert a single row with the specified expression 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, the 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 updateable and has only one table in the FROM clause.

An inherently non-updateable 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 lower case. 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. For more information, see "ALTER TABLE statement" on page 294.

Permissions Must have INSERT permission on the table.

Side effects None.

See also

- ♦ “INPUT statement [Interactive SQL]” on page 523
- ♦ “LOAD TABLE statement” on page 538
- ♦ “UPDATE statement” on page 650
- ♦ “DELETE statement” on page 445
- ♦ “PUT statement [ESQL]” on page 567

Standards and compatibility

- ♦ **SQL/92** Entry-level feature. INSERT … ON EXISTING is a vendor extension.
- ♦ **SQL/99** Core feature. INSERT … ON EXISTING is a vendor extension.
- ♦ **Sybase** Supported by Adaptive Server Enterprise.

Examples

Add an Eastern Sales department to the database.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 230, ’Eastern Sales’ )
```

Create the table dept_head and fill it with the names of department heads and their departments.

```
CREATE TABLE dept_head(
      pk int primary key default autoincrement,
      dept_name varchar(128),
      manager_name varchar (128) );
INSERT
INTO dept_head (manager_name, dept_name)
SELECT emp_fname || ’ ’ || emp_lname, dept_name
FROM employee JOIN department
ON emp_id = dept_head_id
```

Create the table dept_head and fill it with the names of department heads and their departments using the WITH AUTO NAME syntax.

```
CREATE TABLE dept_head(
      pk int primary key default autoincrement,
      dept_name varchar(128),
      manager varchar (128) );
INSERT
INTO dept_head WITH AUTO NAME
SELECT emp_fname || ’ ’ || emp_lname AS manager,
      dept_name
FROM employee JOIN department
ON emp_id = dept_head_id
```

Create the table mytab and populate it using the WITH AUTO NAME syntax.

```
CREATE TABLE mytab(
      pk int primary key default autoincrement,
      table_name char(128),
      len int );
INSERT into mytab WITH AUTO NAME
SELECT
      length(t.table_name) AS len,
      t.table_name
FROM SYS.SYSTABLE t
WHERE table_id<=10
```

INSTALL JAVA statement

Description

Use this statement to make Java classes available for use within a database.

Syntax

INSTALL JAVA
[**NEW** | **UPDATE**]
[**JAR** *jar-name*]
FROM { **FILE** *filename* | *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 *filename* 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 *filename* 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.

Usage

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 (for example, with a STOP JAVA and START JAVA).

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 576

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

The following statement installs the user-created Java class named Demo, by providing the filename and location of the class.

```
INSTALL JAVA NEW
FROM FILE 'D:\JavaClass\Demo.class'
```

After installation, the class is referenced using its name. Its original file path location is no longer used. For example, the following statement uses the class installed in the previous statement.

```
create variable d Demo
```

If the Demo class was a member of the package *sybase.work*, the fully qualified name of the class must be used, for example,

```
CREATE VARIABLE d sybase.work.Demo
```

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 operation

Description

Computes the intersection between the result sets of two or more queries.

Syntax

select-statement
  INTERSECT [**ALL** | **DISTINCT**] *select-statement*
[**INTERSECT** [**ALL** | **DISTINCT**] *select-statement*] . . .
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], . . .]

Usage

The intersection between the result sets of several SELECT statements can be obtained as a single result using INTERSECT or INTERSECT ALL. INTERSECT DISTINCT is identical to INTERSECT.

The component SELECT statements must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

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, Adaptive Server Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each component SELECT statement 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 SELECT statement. An alternative way of customizing result set column names is to use the WITH clause on the SELECT statement.

Permissions

Must have SELECT permission for each of the component SELECT statements.

Side effects

None.

See also

"EXCEPT operation" on page 469

"UNION operation" on page 644

Standards and compatibility

- **SQL/92** Entry-level.
- **SQL/99** Feature F302.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

For examples of INTERSECT usage, see "Set operators and NULL" [*ASA SQL User's Guide*, page 256].

LEAVE statement

Description

Use this statement to leave a compound statement or loop.

Syntax

LEAVE *statement-label*

See also

- "LOOP statement" on page 548
- "FOR statement" on page 487
- "BEGIN statement" on page 312
- "Using Procedures, Triggers, and Batches" [*ASA SQL User's Guide*, page 655]

Usage

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.

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Not supported in Adaptive Server Enterprise. The BREAK statement provides a similar feature for Transact-SQL compatible procedures.

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

Description

This statement loads statistics into the system table SYSCOLSTAT. 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 SYSCOLSTAT 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

- "SYSCOLSTAT system table" on page 688
- "Unloading a database using the dbunload command-line utility" [*ASA Database Administration Guide*, page 590]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

LOAD TABLE statement

Description

Use this statement to import bulk data into a database table from an external ASCII-format file. *Inserts are not recorded in the log file*, raising the risk that data will be lost in the event of a crash and making this statement unusable with SQL Remote or with MobiLink remote databases.

Syntax

LOAD [**INTO**] **TABLE** [*owner.*]*table-name* [**(** *column-name*, . . . **)**]
FROM *filename*
[*load-option* . . .]
[*statistics-limitation-options*]

load-option :
  CHECK CONSTRAINTS { **ON** | **OFF** }
| **COMPUTES** { **ON** | **OFF** }
| **DEFAULTS** { **ON** | **OFF** }
| **DELIMITED BY** *string*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **ASCII** | **BCP** }
| **HEXADECIMAL** {**ON** | **OFF**}
| **ORDER** {**ON** | **OFF**}
| **PCTFREE** *percent-free-space*
| **QUOTES** { **ON** | **OFF** }
| **SKIP** *integer*
| **STRIP** { **ON** | **OFF** }
| **WITH CHECKPOINT** { **ON** | **OFF** }

statistics-limitation-options :
STATISTICS { **ON** [**ALL COLUMNS**] | **OFF** | **ON KEY COLUMNS** | **ON (** *column-list*
) }

filename : { *string* | *variable* }

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 will be used. If DEFAULTS is off and a non-nullable column is omitted from the column list, the engine 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 will be 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 *filename-string* is passed to the server as a string. The string is therefore 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 employee table is:

  ```
  LOAD TABLE employee
  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 server machine, not on the client machine.

- ♦ You can use UNC path names to load data from files on computers other than the server. For example, on a Windows 95 or Windows NT network, you may use the following statement to load data from a file on the client machine:

  ```
  LOAD TABLE employee
  FROM '\\\\client\\temp\\input.dat'
  ```

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.

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 character is a comma. You can specify an alternative column delimiter by providing a string. The same formatting requirements apply as to other SQL strings. In particular, if you wanted to specify tab-delimited values, the hexadecimal ASCII code of the tab character (9) is used. The DELIMITED BY clause is as follows:

```
...DELIMITED BY '\x09' ...
```

You can specify delimiters that are up to 255 bytes in length. For example,

```
...DELIMITED BY '###' ...
```

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 character. Choosing BCP allows the import of ASE 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 multi-byte 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.

☞ For more information, see "Using clustered indexes" [*ASA SQL User's Guide*, page 63].

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 characters can be included in column values. Also, quote characters are assumed not to be part of the value. Therefore, a line of the form

```
'123 High Street, Anytown',(715)398-2354
```

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.

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',''''
```

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 with a quoted string.

STRIP option With STRIP turned on (the default), trailing blanks are stripped from values before they are inserted. To turn the STRIP option off, the clause is as follows:

```
...STRIP OFF ...
```

Trailing blanks are stripped only for non-quoted strings. Quoted strings retain their trailing blanks. Leading blanks are trimmed, regardless of the STRIP setting, unless they are enclosed in quotes.

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.*

☞ For more information, see CONVERSION_ERROR option.

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.

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.

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.

☞ For more information about PCTFREE, see "CREATE TABLE statement" on page 407.

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.

Usage

> **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.*

The LOAD TABLE statement allows efficient mass insertion into a database table from an ASCII file. LOAD TABLE is more efficient than the Interactive SQL statement INPUT. Before inserting data, you can specify the percentage of each table page that should be left free for later updates. For more information, see the "ALTER TABLE statement" on page 294.

LOAD TABLE places an exclusive lock on the whole table.

You can use LOAD TABLE on temporary tables, but the temporary table must have been created with the ON COMMIT PRESERVE ROWS clause because LOAD TABLE does a COMMIT after the load.

If the ASCII file has entries such that a column appears to be NULL, LOAD TABLE treats it as null. If the column in that position cannot be NULL, it inserts a zero in numeric columns and an empty string in character columns. LOAD TABLE skips empty lines in the input file.

LOAD TABLE and statistics LOAD TABLE captures column statistics when it loads data in order to create histograms on table columns. If a histogram already exists for a column, LOAD TABLE leaves the existing histogram alone and does not create a new one. If you are loading into an empty table, it may be beneficial to drop statistics first.

LOAD TABLE does not generate statistics for columns that contain NULL values for more than 90% of the rows being loaded.

LOAD TABLE saves statistics on base tables for future use. It does not save statistics on global temporary tables.

LOAD TABLE adds statistics only if the table has five or more rows. If the table has at least five rows, histograms are added as follows:

Data already in table?	Histogram present?	Action taken
Yes	Yes	Update existing histograms
Yes	No	Don't build histograms
No	Yes	Update existing histograms
No	No	Build new histograms

☞ For more information, see “Optimizer estimates and histograms” [*ASA SQL User's Guide*, page 397].

Using dynamically constructed filenames You can execute a LOAD TABLE statement with a dynamically constructed filename by dynamically constructing the entire statement, then executing it using the EXECUTE IMMEDIATE statement. For more information, see the “EXECUTE IMMEDIATE statement [SP]” on page 475.

Permissions

The permissions required to execute a LOAD TABLE statement depend on the database server `-gl` command line option, as follows:

♦ If the -gl option is ALL, you must be the owner of the table or have DBA authority or have ALTER privilege.

- If the -gl option is DBA, you must have DBA authority.
- If the -gl option is NONE, LOAD TABLE is not permitted.

☞ For more information, see “-gl server option” [*ASA Database Administration Guide*, page 144].

Requires an exclusive lock on the table.

Side effects

Inserts are not recorded in the log file. Thus, the inserted rows may not be recovered in the event of a crash. 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.

Side effects

Automatic commit.

See also

- “UNLOAD TABLE statement” on page 648
- “INSERT statement” on page 528
- “INPUT statement [Interactive SQL]” on page 523

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

Following is an example of LOAD TABLE. First, we create a table, 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 filename, via the EXECUTE IMMEDIATE statement:

```
CREATE PROCEDURE LoadData( IN from_file LONG VARCHAR )
BEGIN
    DECLARE cmd LONG VARCHAR;
    SET cmd = ’LOAD TABLE DBA.MyTable FROM ’ ||
      ’’’d:\\data\\’ || from_file || ’’’’;
    EXECUTE IMMEDIATE WITH ESCAPES OFF cmd;
END
```

LOCK TABLE statement

Description

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 release when the current transaction is committed or rolled back.

SHARE mode Prevent other transactions from modifying the table, but allow them read access. In this mode you can change data in the table as long as no other transaction has locked the row being modified, either indirectly or explicitly using LOCK TABLE.

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 **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 OFF.

Usage

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.

The locks placed by LOCK TABLE in SHARE mode are phantom and anti-phantom locks, which are displayed by the sa_locks procedure as PT and AT.

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 597
- "sa_locks system procedure" on page 805

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported in Adaptive Server Enterprise. The WITH HOLD clause is not supported in Adaptive Server Enterprise. Adaptive Server Enterprise provides a WAIT clause that is not supported in Adaptive Server Anywhere.

Example

The following statement prevents other transactions from modifying the customer table for the duration of the current transaction:

```
LOCK TABLE customer
IN SHARE MODE
```

LOOP statement

Description

Use this statement to repeat the execution of a statement list.

Syntax

[*statement-label* :]
[**WHILE** *search-condition*] **LOOP**
  statement-list
END LOOP [*statement-label*]

Usage

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 487
- "LEAVE statement" on page 535

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Not supported in Adaptive Server Enterprise. The WHILE statement provides looping in Transact-SQL stored procedures.

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

Description

Use this statement to display a message.

Syntax

```
MESSAGE expression, . . .
[ TYPE { INFO | ACTION | WARNING | STATUS } ]
[ TO { CONSOLE
   | CLIENT [ FOR { CONNECTION conn_id | ALL } ]
   | LOG }
[ DEBUG ONLY ]
]
```

conn_id : integer

Parameters

TYPE clause The TYPE clause only has an effect if the message is sent to the client. The client application must decide how to handle the message. Interactive SQL displays messages in the following locations:

- ♦ **INFO** The Messages pane. INFO is the default type.
- ♦ **ACTION** A Message box with an OK button.
- ♦ **WARNING** A Message box with an OK button.
- ♦ **STATUS** The Messages pane.

TO clause This clause specifies the destination of a message:

- ♦ **CONSOLE** Send messages to the database server 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.

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.

Usage

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 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.

ESQL and ODBC clients receive messages via message callback functions. In each case, these functions must be registered. To register ESQL message handlers, use the db_register_callback function.

ODBC clients can register callback functions using the SQLSetConnectAttr function.

Permissions

DBA authority is required to execute a MESSAGE statement containing a FOR clause.

Side effects

None.

See also

- "CREATE PROCEDURE statement" on page 373
- "DEBUG_MESSAGES option [database]" [*ASA Database Administration Guide*, page 648]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise. The Transact-SQL PRINT statement provides a similar feature, and is also available in Adaptive Server Anywhere.

Examples

1. The following procedure displays a message on the server message window:

   ```
   CREATE PROCEDURE message_test ()
   BEGIN
   MESSAGE 'The current date and time: ', Now();
   END
   ```

   The statement:

   ```
   CALL message_test()
   ```

   displays the string *The current date and time*, and the current date and time, on the database server message window.

2. To register a callback in ODBC, first declare the message handler:

   ```
   void SQL_CALLBACK my_msgproc(
      void *     sqlca,
      unsigned char     msg_type,
      long              code,
      unsigned short    len,
      char*             msg )
   { ... }
   ```

Install the declared message handler by calling the SQLSetConnectAttr function.

```
rc = SQLSetConnectAttr(
   dbc,
   ASA_REGISTER_MESSAGE_CALLBACK,
   (SQLPOINTER) &my_msgproc, SQL_IS_POINTER );
```

OPEN statement [ESQL] [SP]

Description

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 will be 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]" [*ASA Database Administration Guide*, page 689]), 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 will be 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" [*ASA SQL User's Guide*, page 135].

The cursor is positioned before the first row (see "Using cursors in embedded SQL" [*ASA Programming Guide*, page 167] or "Using cursors in procedures and triggers" [*ASA SQL User's Guide*, page 692]).

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.

☞ For more information, see "FETCH statement [ESQL] [SP]" on page 482.

Usage

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 436
- "RESUME statement" on page 582
- "PREPARE statement [ESQL]" on page 563
- "FETCH statement [ESQL] [SP]" on page 482
- "RESUME statement" on page 582
- "CLOSE statement [ESQL] [SP]" on page 325

Standards and compatibility

- **SQL/92** Embedded SQL use is an entry-level feature. Procedures use is a Persistent Stored Modules feature.

- **SQL/99** Embedded SQL use is a core feature. Procedures use is a Persistent Stored Modules feature.

- **Sybase** The simple OPEN *cursor-name* syntax is supported by Adaptive Server Enterprise. None of the other clauses are supported in Adaptive Server Enterprise stored procedures. Open Client/Open Server supports the USING descriptor or host variable syntax.

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 employee 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 emp_lname
   FROM employee;
DECLARE name CHAR(40);
OPEN cur_employee;
LOOP
FETCH NEXT cur_employee into name;
    ...
END LOOP
CLOSE cur_employee;
END
```

OUTPUT statement [Interactive SQL]

Description

Use this statement to output the current query results to a file.

Syntax

OUTPUT TO *filename*
[**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 will be 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 '!'
```

ESCAPES clause 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.

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.

If *encoding* is not specified, Interactive SQL determines the code page that is used to write the file as follows, where code page values occurring earlier in the list take precedence over those occurring later in the list:

♦ the code page specified with the DEFAULT_ISQL_ENCODING option (if this option is set)

♦ the code page specified with the -codepage option when Interactive SQL was started

♦ the default code page for the computer Interactive SQL is running on

☞ For more information about Interactive SQL and encodings, see "DEFAULT_ISQL_ENCODING option [Interactive SQL]" [*ASA Database Administration Guide,* page 649].

Usage

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]" [*ASA Database Administration Guide,* page 679]).

The current query is the SELECT or INPUT statement which 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.

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 597
- "INPUT statement [Interactive SQL]" on page 523
- "UNLOAD TABLE statement" on page 648

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Examples

Place the contents of the employee table in a file in ASCII format:

```
SELECT *
FROM employee;
OUTPUT TO employee.txt
FORMAT ASCII
```

Place the contents of the employee table at the end of an existing file, and include any messages about the query in this file as well:

```
SELECT *
FROM employee;
OUTPUT TO employee.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]

Description	Use this statement to specify parameters to an Interactive SQL command file.
Syntax	**PARAMETERS** *parameter1*, *parameter2*, . . .

Usage

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 wish 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 571

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

The following Interactive SQL command file takes two parameters.

```
PARAMETERS department_id, file;
SELECT emp_lname
FROM employee
WHERE dept_id = {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]

Description	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**

Usage

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.

See also

♦ "sp_passthrough procedure" [*SQL Remote User's Guide*, page 402]

Example

```
PASSTHROUGH FOR rem_db ;
...
( SQL statements to be executed at the remote database )
...
PASSTHROUGH STOP ;
```

PREPARE statement [ESQL]

Description

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.

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, we recommend 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.

Usage

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 436
- "DESCRIBE statement [ESQL]" on page 449
- "OPEN statement [ESQL] [SP]" on page 553
- "EXECUTE statement [ESQL]" on page 471
- "DROP STATEMENT statement [ESQL]" on page 463

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

Example

The following statement prepares a simple query:

```
EXEC SQL PREPARE employee_statement FROM
'SELECT emp_lname FROM employee';
```

PREPARE TO COMMIT statement

Description	Use this statement to check whether a COMMIT can be performed successfully.
Syntax	**PREPARE TO COMMIT**
Usage	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.
Permissions	None.
Side effects	None.
See also	♦ "COMMIT statement" on page 329 ♦ "ROLLBACK statement" on page 591
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not supported in Adaptive Server Enterprise.

Example

The following sequence of statements leads to an error because of foreign key checking on the employee table.

```
EXECUTE IMMEDIATE
   "SET OPTION wait_for_commit = 'on'";
EXECUTE IMMEDIATE "DELETE FROM employee
   WHERE emp_id = 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 department
WHERE dept_id = 100;
PREPARE TO COMMIT;
```

PRINT statement [T-SQL]

Description

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*]

Usage

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 database server 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 549

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

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]

Description

Use this statement to insert a row into the specified cursor.

Syntax

PUT *cursor-name*
[**USING DESCRIPTOR** *sqlda-name* | **FROM** *hostvar-list*]
[**INTO** { **DESCRIPTOR** *into-sqlda-name* | *into-hostvar-list* }]
[**ARRAY** :*nnn*]

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

hostvar-list : may contain indicator variables

Usage

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 updateable 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 will be used; otherwise, a NULL value will be 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 604.

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. For more information, see "Modifying rows through a cursor" [*ASA Programming Guide*, page 23].

See also

- "UPDATE statement" on page 650
- "UPDATE (positioned) statement [ESQL] [SP]" on page 655
- "DELETE statement" on page 445
- "DELETE (positioned) statement [ESQL] [SP]" on page 447
- "INSERT statement" on page 528

Standards and compatibility

- **SQL/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

Example

The following statement illustrates the use of PUT in Embedded SQL:

```
EXEC SQL PUT cur_employee FROM :emp_id, :emp_lname;
```

RAISERROR statement [T-SQL]

Description

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 Adaptive Server 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 SYS.SYSUSERMESSAGES table.

In Adaptive Server 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 Adaptive Server 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.

Usage

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 426
- "ON_TSQL_ERROR option [compatibility]" [*ASA Database Administration Guide*, page 675]
- "CONTINUE_AFTER_RAISERROR option [compatibility]" [*ASA Database Administration Guide*, page 644]

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

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]" [*ASA Database Administration Guide*, page 666].

READ statement [Interactive SQL]

Description

Use this statement to read Interactive SQL statements from a file.

Syntax

READ [**ENCODING** *encoding*] *filename* [*parameters*]

encoding : *identifier* or *string*

Usage

The READ statement reads a sequence of Interactive SQL statements from the named file. This file can contain any valid Interactive SQL statement, 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 code page that is used to read the file as follows, where code page values occurring earlier in the list take precedence over those occurring later in the list:

♦ the code page specified with the DEFAULT_ISQL_ENCODING option (if this option is set)

♦ the code page specified with the -codepage option when Interactive SQL was started

♦ the default code page for the computer Interactive SQL is running on

☞ For more information about Interactive SQL and encodings, see "DEFAULT_ISQL_ENCODING option [Interactive SQL]" [*ASA Database Administration Guide*, page 649].

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 561). 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 561

Standards and compatibility

♦ **SQL/92** Vendor extension.

♦ **SQL/99** Vendor extension.

♦ **Sybase** Not applicable.

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]

Description

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**]

Usage

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 668
- "GET DATA statement [ESQL]" on page 497
- "TEXTPTR function [Text and image]" on page 234

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

  Adaptive Server Enterprise supports the following clause, which is not supported by Adaptive Server Anywhere:

  ```
  USING { BYTES | CHARS | CHARACTERS }
  ```

  These options are identical for all single-byte character sets. Adaptive Server Anywhere uses **bytes** only, which is the Adaptive Server Enterprise default setting.

  Adaptive Server Enterprise also provides isolation level control in the READTEXT statement. This is not supported in Adaptive Server Anywhere.

RELEASE SAVEPOINT statement

Description	Use this statement to release a savepoint within the current transaction.
Syntax	**RELEASE SAVEPOINT** [*savepoint-name*]

Usage

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" on page 315
- ♦ "COMMIT statement" on page 329
- ♦ "ROLLBACK statement" on page 591
- ♦ "ROLLBACK TO SAVEPOINT statement" on page 592
- ♦ "SAVEPOINT statement" on page 596
- ♦ "Savepoints within transactions" [*ASA SQL User's Guide,* page 104]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not supported by Adaptive Server Enterprise. A similar feature is available in an Adaptive Server Enterprise-compatible manner using nested transactions.

REMOTE RESET statement [SQL Remote]

Description

Use this statement in custom database-extraction procedures to start all subscriptions for a remote user in a single transaction.

Syntax

REMOTE RESET *userid*

Usage

This command starts all subscriptions for a remote user in a single transaction. It sets the **log_sent** and **confirm_sent** values in **SYSREMOTEUSER** table to the current position in the transaction log. It also sets the created and started values in **SYSSUBSCRIPTION** 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.

In order 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 628

Example

- The following statement resets the subscriptions for remote user SamS:

```
REMOTE RESET SamS
```

REMOVE JAVA statement

Description

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* [, *java_class_name*, . . .]
| **JAR** *jar_name* [, *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.

Usage

Removes a class or jar file from the database.

Permissions

Must have DBA authority.

Not supported on Windows CE.

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. A similar feature is available in an Adaptive Server Enterprise-compatible manner using nested transactions.

Example

The following statement removes a Java class named Demo from the current database.

```
REMOVE JAVA CLASS Demo
```

REORGANIZE TABLE statement

Description

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" [*ASA SQL User's Guide*, page 63]

Usage

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 the group is being processed; once the 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. This form of the statement can only be used with databases created with Adaptive Server Anywhere version 7.0 or above. 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. For version 7.0 or earlier databases, this may result in growth of the database file. For version 8.0 or later databases, this will result in only a temporary increase in the database file size, since the checkpoint log is deleted at shutdown and the file is truncated at that point. Also, more contiguous allocation of table pages may result for version 8.0 or later databases.

Neither form of the statement is 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.

Side effects

Prior to starting the reorganization, a checkpoint is done to try to maximize the number of free pages.

Examples

The following statement reorganizes the primary key index for the employee table:

```
REORGANIZE TABLE employee
PRIMARY KEY
```

The following statement reorganizes the table pages of the employee table:

```
REORGANIZE TABLE  employee
```

The following statement reorganizes the index ix_prod_name on the product table:

```
REORGANIZE TABLE product
   INDEX ix_prod_name
```

The following statement reorganizes the foreign key ky_dept_id for the employee table:

```
REORGANIZE TABLE employee
   FOREIGN KEY ky_dept_id
```

RESIGNAL statement

Description

Use this statement to resignal an exception condition.

Syntax

RESIGNAL [*exception-name*]

Usage

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 will be 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 622
- ♦ "BEGIN statement" on page 312
- ♦ "Using exception handlers in procedures and triggers" [*ASA SQL User's Guide*, page 700]
- ♦ "RAISERROR statement [T-SQL]" on page 569

Standards and compatibility

- ♦ **SQL/92** Persistent stored module feature.
- ♦ **SQL/99** Persistent Stored Module feature.
- ♦ **Sybase** Not supported in Adaptive Server Enterprise. Signaling of errors in Transact-SQL procedures is carried out using the RAISERROR statement.

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

Description

Use this statement to restore a backed up database from an archive.

Syntax

RESTORE DATABASE *filename*
FROM *archive-root*
[**CATALOG ONLY |**
[[**RENAME** *dbspace-name* **TO** *new-dbspace-name*] . . .]]
[**HISTORY** { **ON** | **OFF** }]

filename : { *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

Usage

Each RESTORE DATABASE operation updates a history file called *backup.syb*.

☞ For more information about the *backup.syb* file, see “ASLOGDIR environment variable” [*ASA Database Administration Guide,* page 278].

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.

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.

This statement is not supported on Windows CE.

☞ For more information, see “-gu server option” [*ASA Database Administration Guide*, page 148].

Side effects

None.

See also

- “BACKUP statement” on page 307
- “Backup and Data Recovery” [*ASA Database Administration Guide*, page 373]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported in Adaptive Server Enterprise.
- **Windows CE** Not supported on the Windows CE platform.

Example

The following example restores a database from a Windows NT 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 ON, but is OFF in utility_db. When connected to any database other than utility_db, the extra backslashes are required.

```
RESTORE DATABASE 'd:\\dbhome\\cust.db'
FROM '\\\\.\\tape0'
```

RESUME statement

Description

Use this statement to resume execution of a cursor that returns result sets.

Syntax

RESUME *cursor-name*

cursor-name : *identifier* or *hostvar*

Usage

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.

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 436
- "Returning results from procedures" [*ASA SQL User's Guide*, page 686]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise.

Example

Following are Embedded SQL examples.

```
1. EXEC SQL RESUME cur_employee;
2. EXEC SQL RESUME :cursor_var;
```

Following is an Interactive SQL example.

```
CALL sample_proc();
RESUME ALL;
```

RETURN statement

Description

Use this statement to exit from a function or procedure unconditionally, optionally providing a return value.

Syntax

RETURN [*expression*]

Usage

A RETURN statement causes an immediate exit from a function or procedure. If *expression* is supplied, the value of *expression* is returned as the value of the function or procedure.

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 362
- "CREATE PROCEDURE statement" on page 373
- "BEGIN statement" on page 312

Standards and compatibility

- **SQL/92** Persistent stored module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Transact-SQL procedures use the RETURN statement to return an integer error code.

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 customer)
   OR customer_id IS NULL THEN
      RETURN
   ELSE
      SELECT product.id,sum(
         sales_order_items.quantity )
      FROM  product,
            sales_order_items,
            sales_order
      WHERE sales_order.cust_id=customer_id
      AND sales_order.id=sales_order_items.id
      AND sales_order_items.prod_id=product.id
      GROUP BY product.id
   END IF
END
```

REVOKE statement

Description

Use this statement to remove permissions for the specified users.

Syntax 1

REVOKE *special-priv*, . . . **FROM** *userid*, . . .

special-priv :
 CONNECT
| **DBA**
| **INTEGRATED LOGIN**
| **GROUP**
| **MEMBERSHIP IN GROUP** *userid*, . . .
| **RESOURCE**

Syntax 2

REVOKE *table-priv*, . . .
ON [*owner*.]*table-name*
FROM *userid*, . . .

table-priv :
 ALL [**PRIVILEGES**]
| **ALTER**
| **DELETE**
| **INSERT**
| **REFERENCES** [**(** *column-name*, . . . **)**]
| **SELECT** [**(** *column-name*, . . . **)**]
| **UPDATE** [**(** *column-name*, . . . **)**]

Syntax 3

REVOKE EXECUTE
ON [*owner*.]*procedure-name*
FROM *userid*, . . .

Usage

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, etc.) owned by that user and any permissions granted by that user. REVOKE GROUP automatically REVOKES MEMBERSHIP from all members of the group.

When you add a user to a group, the user inherits all the permissions assigned to that group. Adaptive Server 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 individually to each individual user ID that requires them.

If you give a user GRANT option permission, and later revoke that permission, you also revoke any permissions that that user granted to others while they had the GRANT option.

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.

Side effects

Automatic commit.

See also

- "GRANT statement" on page 503

Standards and compatibility

- **SQL/92** Syntax 1 is a vendor extension. Syntax 2 is an entry-level feature. Syntax 3 is a Persistent Stored Modules feature.
- **SQL/99** Syntax 1 is a vendor extension. Syntax 2 is a core feature. Syntax 3 is a Persistent Stored Modules feature.
- **Sybase** Syntax 2 and 3 are supported by Adaptive Server Enterprise. Syntax 1 is not supported by Adaptive Server Enterprise. User management and security models are different for Adaptive Server Anywhere and Adaptive Server Enterprise.

Example

Prevent user Dave from updating the employee table.

```
REVOKE UPDATE ON employee FROM dave;
```

Revoke resource permissions from user Jim.

```
REVOKE RESOURCE FROM Jim;
```

Revoke integrated login mapping from user profile name Administrator.

```
REVOKE INTEGRATED LOGIN FROM Administrator;
```

Disallow the Finance group from executing the procedure sp_customer_list.

```
REVOKE EXECUTE ON sp_customer_list
FROM finance;
```

Drop user ID FranW from the database.

```
REVOKE CONNECT FROM FranW
```

REVOKE CONSOLIDATE statement [SQL Remote]

Description	Use this statement to stop a consolidated database from receiving SQL Remote messages from this database.
Syntax	**REVOKE CONSOLIDATE FROM** *userid*
Usage	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 588
- "REVOKE REMOTE statement [SQL Remote]" on page 589
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 590
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 508
- "sp_revoke_consolidate procedure" [*SQL Remote User's Guide*, page 431]

Example

- The following statement revokes consolidated status from the user ID **condb**:

```
REVOKE CONSOLIDATE FROM condb
```

REVOKE PUBLISH statement [SQL Remote]

Description

Use this statement to terminate the identification of the named user ID as the CURRENT publisher.

Syntax

REVOKE PUBLISH FROM *userid*

Usage

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 510
- ♦ “REVOKE REMOTE statement [SQL Remote]” on page 589
- ♦ “REVOKE REMOTE DBA statement [SQL Remote]” on page 590
- ♦ “REVOKE CONSOLIDATE statement [SQL Remote]” on page 587
- ♦ “sp_publisher procedure” [*SQL Remote User’s Guide*, page 409]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.

Example

```
REVOKE PUBLISH FROM publisher_ID
```

REVOKE REMOTE statement [SQL Remote]

Description	Use this statement to stop a user from being able to receive SQL Remote messages from this database.
Syntax	**REVOKE REMOTE FROM** *userid*, . . .
Usage	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 588 ♦ "GRANT REMOTE statement [SQL Remote]" on page 511 ♦ "REVOKE REMOTE DBA statement [SQL Remote]" on page 590 ♦ "REVOKE CONSOLIDATE statement [SQL Remote]" on page 587 ♦ "sp_revoke_remote procedure" [*SQL Remote User's Guide*, page 432]
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension.
Example	`REVOKE REMOTE FROM SamS`

REVOKE REMOTE DBA statement [SQL Remote]

Description	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*, . . .
Usage	REMOTE DBA authority enables the Message Agent to have full access to the database in order 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 588 ♦ “REVOKE REMOTE statement [SQL Remote]” on page 589 ♦ “GRANT REMOTE DBA statement [SQL Remote]” on page 513 ♦ “REVOKE CONSOLIDATE statement [SQL Remote]” on page 587 ♦ “The Message Agent and replication security” [*SQL Remote User’s Guide*, page 243]
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension.

ROLLBACK statement

Description	Use this statement to end a transaction and undo any changes made since the last COMMIT or ROLLBACK.
Syntax	**ROLLBACK** [**WORK**]
Usage	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 329 ♦ "ROLLBACK TO SAVEPOINT statement" on page 592
Standards and compatibility	♦ **SQL/92** Entry-level feature. ♦ **SQL/99** Core feature. ♦ **Sybase** Supported by Adaptive Server Enterprise.

ROLLBACK TO SAVEPOINT statement

Description

To cancel any changes made since a SAVEPOINT.

Syntax

ROLLBACK TO SAVEPOINT [*savepoint-name*]

Usage

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" on page 315
- "COMMIT statement" on page 329
- "RELEASE SAVEPOINT statement" on page 574
- "ROLLBACK statement" on page 591
- "SAVEPOINT statement" on page 596
- "Savepoints within transactions" [*ASA SQL User's Guide*, page 104]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Savepoints are not supported by Adaptive Server Enterprise. To implement similar features in an Adaptive Server Enterprise-compatible manner, you can use nested transactions.

  ☞ For more information on nested transactions, see "BEGIN TRANSACTION statement" on page 315.

ROLLBACK TRANSACTION statement

Description	Use this statement to cancel any changes made since a SAVE TRANSACTION.
Syntax	**ROLLBACK TRANSACTION** [*savepoint-name*]

Usage

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 592
- "BEGIN TRANSACTION statement" on page 315
- "COMMIT statement" on page 329"SAVE TRANSACTION statement" on page 595

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Supported by Adaptive Server Enterprise.

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

Description

Use this statement to undo any changes made in a trigger.

Syntax

ROLLBACK TRIGGER [**WITH** *raiserror-statement*]

Usage

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 419
- "ROLLBACK statement" on page 591
- "ROLLBACK TO SAVEPOINT statement" on page 592
- "RAISERROR statement [T-SQL]" on page 569

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

SAVE TRANSACTION statement

Description

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVE TRANSACTION *savepoint-name*

Usage

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.

☞ For more information, see "Savepoints within transactions" [*ASA SQL User's Guide*, page 104].

Permissions

None.

Side effects

None.

See also

- ♦ "SAVEPOINT statement" on page 596
- ♦ "BEGIN TRANSACTION statement" on page 315
- ♦ "COMMIT statement" on page 329
- ♦ "ROLLBACK TRANSACTION statement" on page 593

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Supported by Adaptive Server Enterprise.

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

Description

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVEPOINT [*savepoint-name*]

Usage

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" [*ASA SQL User's Guide*, page 104].

Savepoints that are established while a trigger or atomic compound statement is executing are automatically released when the atomic operation ends.

Permissions

None.

Side effects

None.

See also

- "RELEASE SAVEPOINT statement" on page 574
- "ROLLBACK TO SAVEPOINT statement" on page 592

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Not supported in Adaptive Server Enterprise. To implement similar features in an Adaptive Server Enterprise-compatible manner, you can use the SAVE TRANSACTION statement.

SELECT statement

Description

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** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** [*group-by-expression*]
[**HAVING** *search-condition*]
[**WINDOW** *window-name* **AS** *window-spec*
    [, *window-name* **AS** *window-spec* ...]]
[**ORDER BY** { *expression* | *integer* } [**ASC** | **DESC**], . . .]
[**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
[**FOR XML** *xml-mode*]

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 :
{ *column-name* | *expression* } [[**AS**] *alias-name*], . . .
| *****| *window-function* **OVER** { *window-name* | *window-spec* }

group-by-expression:

See “GROUP BY clause” on page 514.

cursor-concurrency :
  BY { **VALUES** | **TIMESTAMP** | **LOCK** }

xml-mode :
RAW [, **ELEMENTS**] | **AUTO** [, **ELEMENTS**] | **EXPLICIT**
window-function :**RANK ( )**
| **DENSE_RANK ( )**
| **PERCENT_RANK ( )**
| **CUME_DIST ( )**
| **ROW_NUMBER ( )**
| *aggregate-function*

window-spec:
([*window-name*]
[**PARTITION BY***column-reference* [*collate-clause*]
[, *column-reference* [*collate-clause*] ...]]
[**ORDER BY** *order-by-expression*, ...]
[{ **ROWS** | **RANGE** } { *window-frame-start* | *window-frame-between* }]
)

window-frame-start :
{ **UNBOUNDED** | *integer-expression* } **PRECEDING**
| **CURRENT ROW**

window-frame-between :
BETWEEN *window-frame-bound-1*
AND *window-frame-bound-2*

window-frame-bound:
window-frame-start
| { **UNBOUNDED** | *integer-expression* } **FOLLOWING**

Parameters

WITH or WITH RECURSIVE 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.

☞ For more information, see "Common Table Expressions" [*ASA SQL User's Guide,* page 307].

ALL or DISTINCT 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 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" [*ASA SQL User's Guide,* page 251].

select list The select list is a list of expressions, separated by commas, specifying what will be 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 16). 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 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 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 This clause is used to create a table and fill it with data.

If the table name starts # then 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 the following conditions:

- The *select-list* contains more than one item, and the INTO target is a single *table-name* identifier, or
- 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 statement 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,

```
SELECT @@version
```

displays the value of the global variable @@version. This is equivalent to:

```
SELECT @@version
FROM DUMMY
```

☞ For more information, see "FROM clause" on page 491.

WHERE clause This clause specifies which rows will be 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).

☞ For more information, see "Search conditions" on page 23 and "FROM clause" on page 491.

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. All NULL-containing rows are treated as a single set. The resulting rows are often referred to as groups since there is one row in the result for each group of rows from the table list. 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.

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 will be 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, Adaptive Server 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 This clause specifies whether updates are allowed through a cursor opened on the query. Note that this clause cannot be used with the FOR XML clause.

If you do not use a FOR clause in the SELECT statement, the updatability is specified by the API. For ODBC and OLE DB, the default is read only. For JDBC, Open Client, and embedded SQL, the default is update.

This clause overrides the embedded SQL DECLARE CURSOR statement. However, it may be overridden by the concurrency setting in other programming interfaces. In ODBC and OLE DB, the read-only default setting overrides the FOR clause, but if you change the default to something other than read only, the FOR clause is not overridden. In JDBC and Open Client, the current setting always overrides the FOR clause, whether or not it is the default (updatable cursors).

Statement updatability is dependent on the setting of the ANSI_UPDATE_CONSTRAINTS database option. Other characteristics of the statement are also considered, including whether the statement contains a DISTINCT, GROUP BY, HAVING, UNION, aggregate functions, joins, or non-updatable views.

☞ For more information about cursor sensitivity, see "Adaptive Server Anywhere cursors" [*ASA Programming Guide,* page 30].

☞ For more information about ODBC concurrency, see the discussion of SQLSetStmtAttr in "Choosing a cursor characteristics" [*ASA Programming Guide,* page 249].

☞ For more information about the ANSI_UPDATE_CONSTRAINTS database option, see "ANSI_UPDATE_CONSTRAINTS option [compatibility]" [*ASA Database Administration Guide,* page 635].

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. Note that 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.

☞ For information about writing a query using EXPLICIT mode, see "Using FOR XML EXPLICIT" [*ASA SQL User's Guide,* page 533].

☞ For more information about using the FOR XML clause, see "Using the FOR XML clause to retrieve query results as XML" [*ASA SQL User's Guide,* page 525].

Usage

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. The 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.

Permissions

Must have SELECT permission on the named tables and views.

Side effects

None.

See also

- "Expressions" on page 16
- "FROM clause" on page 491
- "Search conditions" on page 23
- "UNION operation" on page 644
- "Joins: Retrieving Data from Several Tables" [*ASA SQL User's Guide*, page 263]

Standards and compatibility

- **SQL/92** Entry-level feature. The complexity of the SELECT statement means that you should check individual clauses against the standard.
- **SQL/99** 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.
- **Sybase** Supported by Adaptive Server Enterprise, with some differences in syntax.

Example

How many employees are there?

```
SELECT count(*)
FROM employee
```

List all customers and the total value of their orders.

```
SELECT company_name,
   CAST( sum(sales_order_items.quantity *
   product.unit_price) AS INTEGER) VALUE
FROM customer
   JOIN sales_order
   JOIN sales_order_items
   JOIN product
GROUP BY company_name
ORDER BY VALUE DESC
```

The following statement shows an Embedded SQL SELECT statement:

```
SELECT count(*) INTO :size
FROM employee
```

SET statement

Description

Use this statement to assign a value to a SQL variable.

Syntax

SET *identifier* = *expression*

Usage

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 427
- "DECLARE statement" on page 435
- "DROP VARIABLE statement" on page 468
- "Expressions" on page 16

Standards and compatibility

- **SQL/92** Persistent stored module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** Not supported. In Adaptive Server Enterprise, variables are assigned using the SELECT statement with no table, a Transact-SQL syntax that is also supported by Adaptive Server Anywhere. The SET statement is used to set database options in Adaptive Server Enterprise.

Example

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]

Description

Use this statement to set database options for the current connection in an Adaptive Server Enterprise-compatible manner.

Syntax

SET *option-name option-value*

Usage

The available options are as follows:

Option name	Option value
ANSINULL	**ON** | **OFF**
ANSI_PERMISSIONS	**ON** | **OFF**
CLOSE_ON_ENDTRANS	**ON** | **OFF**
DATEFIRST	**1** | **2** | **3** | **4** | **5** | **6** | **7**
QUOTED_IDENTIFIER	**ON** | **OFF**
ROWCOUNT	*integer*
SELF_RECURSION	**ON** | **OFF**
STRING_RTRUNCATION	**ON** | **OFF**
TEXTSIZE	*integer*
TRANSACTION ISOLATION LEVEL	**0** | **1** | **2** | **3**

Database options in Adaptive Server Anywhere are set using the SET OPTION statement. However, Adaptive Server 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 Adaptive Server Anywhere as well as in Adaptive Server Enterprise:

- **SET ANSINULL { ON | OFF }** The default behavior for comparing values to NULL in Adaptive Server Anywhere and Adaptive Server Enterprise is different. Setting ANSINULL to OFF provides Transact-SQL compatible comparisons with NULL.

  Adaptive Server Anywhere also supports the following syntax:

  SET ANSI_NULLS { **ON** | **OFF** }

- **SET ANSI_PERMISSIONS { ON | OFF }** The default behavior in Adaptive Server Anywhere and Adaptive Server Enterprise regarding permissions required to carry out an UPDATE or DELETE containing a column reference is different. Setting ANSI_PERMISSIONS to OFF provides Transact-SQL-compatible permissions on UPDATE and DELETE.

- **SET CLOSE_ON_ENDTRANS { ON | OFF }** The default behavior in Adaptive Server Anywhere and Adaptive Server Enterprise for closing cursors at the end of a transaction is different. Setting CLOSE_ON_ENDTRANS to OFF provides Transact-SQL compatible behavior.

- **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]" [*ASA Database Administration Guide,* page 654].

- **SET QUOTED_IDENTIFIER { ON | OFF }** Controls whether strings enclosed in double quotes are interpreted as identifiers (ON) or as literal strings (OFF). For information about this option, see "Setting options for Transact-SQL compatibility" [*ASA SQL User's Guide,* page 485].

- **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 employee set emp_lname='new_name'
      where emp_lname <> '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 Adaptive Server 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 in Adaptive Server Anywhere and Adaptive Server Enterprise when non-space characters are truncated on assigning SQL string data is different. Setting STRING_RTRUNCATION to ON provides Transact-SQL-compatible string comparisons.

- **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 (32K), use the command:

  ```
  set textsize 0
  ```

- **SET TRANSACTION-ISOLATION-LEVEL { 0 | 1 | 2 | 3 }** Sets the locking isolation level for the current connection, as described in "Isolation levels and consistency" [*ASA SQL User's Guide,* page 106]. For Adaptive Server Enterprise, only 1 and 3 are valid options. For Adaptive Server Anywhere, any of 0, 1, 2, or 3 is a valid option.

In addition, the SET statement is allowed by Adaptive Server Anywhere for the PREFETCH option, for compatibility, but has no effect.

Permissions	None.
Side effects	None.

See also

- "SET OPTION statement" on page 613
- "Setting options for Transact-SQL compatibility" [*ASA SQL User's Guide*, page 485]
- "Compatibility options" [*ASA Database Administration Guide*, page 624]

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Adaptive Server Anywhere supports a subset of the Adaptive Server Enterprise database options.

SET CONNECTION statement [Interactive SQL] [ESQL]

Description	Use this statement to change the active database connection.
Syntax	**SET CONNECTION** [*connection-name*] *connection-name* : *identifier*, *string*, or *hostvar*
Usage	The SET CONNECTION statement changes the active database connection to **connection-name**. The current connection state is saved, and will be 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 332 ♦ "DISCONNECT statement [ESQL] [Interactive SQL]" on page 453
Standards and compatibility	♦ **SQL/92** Interactive SQL use is a vendor extension. Embedded SQL is a Full level feature. ♦ **SQL/99** Interactive SQL is a vendor extension. Embedded SQL is a core feature. ♦ **Sybase** Supported by Open Client/Open Server.

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]

Description

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*

Usage

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) will be 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. LONGVARCHAR and LONGBINARY 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 264
- “DEALLOCATE DESCRIPTOR statement [ESQL]” on page 433
- “The SQL descriptor area (SQLDA)” [*ASA Programming Guide*, page 181]

Standards and compatibility

- **SQL/92** Intermediate-level feature.
- **SQL/99** SQL/foundation feature outside of core SQL.
- **Sybase** Supported by Open Client/Open Server.

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 264.

SET OPTION statement

Description

Use this statement to change the values of database options.

Syntax

SET [**EXISTING**] [**TEMPORARY**] **OPTION**
[*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

userid : *identifier* | *string* | *hostvar*

option-name : *identifier* | *string* | *hostvar*

option-value : *hostvar* (indicator allowed)
| *string*
| *identifier*
| *number*

Usage

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.

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" [*ASA Database Administration Guide*, page 613].

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 don't 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 login_mode = mixed
```

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.

In embedded SQL, database options can be set only temporarily.

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 machine. 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 will be 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.

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" [*ASA Database Administration Guide,* page 619]
- "Compatibility options" [*ASA Database Administration Guide,* page 624]
- "Replication options" [*ASA Database Administration Guide,* page 628]
- "SET OPTION statement [Interactive SQL]" on page 616

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Adaptive Server Enterprise. Adaptive Server Anywhere does support some Adaptive Server Enterprise options using the SET statement.

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]

Description

Use this statement to change the values of Interactive SQL options.

Syntax 1

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*

Syntax 2

SET PERMANENT

Syntax 3

SET

Usage

SET PERMANENT (syntax 2) saves all current Interactive SQL options. These settings are automatically established every time Interactive SQL is started for the current user ID.

Syntax 3 displays all of the current option settings. If there are temporary options set for Interactive SQL or the database server, these will be displayed; otherwise, the permanent option settings are displayed.

See also

- “Interactive SQL options” [*ASA Database Administration Guide*, page 630]

SET REMOTE OPTION statement [SQL Remote]

Description

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 | **unlink_delay**

ftp-option:
active_mode | **host** | **password** | **port** | **root_directory** | **user**

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 following locations:

- "The file message system" [*SQL Remote User's Guide*, page 215].
- "The ftp message system" [*SQL Remote User's Guide*, page 216].
- "The MAPI message system" [*SQL Remote User's Guide*, page 220].
- "The SMTP message system" [*SQL Remote User's Guide*, page 218].
- "The VIM message system" [*SQL Remote User's Guide*, page 221].

Usage

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).

Permissions

Must have DBA authority. The publisher can set their own options.

Side effects

Automatic commit.

See also

- "sp_link_option procedure" [*SQL Remote User's Guide*, page 396]
- "Troubleshooting errors at remote sites" [*SQL Remote User's Guide*, page 226]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** 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'
```

SET SQLCA statement [ESQL]

Description

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*

Usage

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" [*ASA Programming Guide*, page 163]). 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" [*ASA Programming Guide*, page 163]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not supported by Open Client/Open Server.

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

Description

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). For more information, see "Identifiers" on page 7 and "Strings" on page 9.

Usage

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 DBA can test the permissions and name resolution of queries, procedures, views, and so on.
- ♦ **Providing a safer environment for administrators** The DBA has permission to carry out any action in the database. If you wish 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 475
- "GRANT statement" on page 503
- "REVOKE statement" on page 585
- "SET OPTION statement" on page 613

Standards and compatibility

- **SQL/92** SET SESSION AUTHORIZATION is SQL 92 compliant. SETUSER is a vendor extension.
- **SQL/99** SET SESSION AUTHORIZATION is a core feature. SETUSER is a vendor extension.
- **Sybase** Adaptive Server Enterprise supports SETUSER, but not the WITH OPTIONS keywords.

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

Description

Use this statement to signal an exception condition.

Syntax

SIGNAL *exception-name*

Usage

SIGNAL allows you to raise an exception. See "Using exception handlers in procedures and triggers" [*ASA SQL User's Guide*, page 700] 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

- "DECLARE statement" on page 435
- "RESIGNAL statement" on page 579
- "BEGIN statement" on page 312
- "Using exception handlers in procedures and triggers" [*ASA SQL User's Guide*, page 700]

Standards and compatibility

- **SQL/92** Persistent Stored Module feature.
- **SQL/99** Persistent Stored Module feature.
- **Sybase** SIGNAL is not supported by Adaptive Server Enterprise.

Example

The following compound statement declares and signals a user-defined exception. If you execute this example from Interactive SQL, the message is displayed in the Messages pane.

```
BEGIN
   DECLARE myexception EXCEPTION
   FOR SQLSTATE '99001';
   SIGNAL myexception;
   EXCEPTION
      WHEN myexception THEN
         MESSAGE 'My exception signaled'
         TO CLIENT;
END
```

START DATABASE statement [Interactive SQL]

Description

Use this statement to start a database on the current database server.

Syntax

START DATABASE *database-file*
[**AS** *database-name*]
[**ON** *engine-name*]
[**WITH TRUNCATE AT CHECKPOINT**]
[**FOR READ ONLY**]
[**AUTOSTOP** { **ON** | **OFF** }]
[**KEY** *key*]

Parameters

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\asademo.db* would be given the default name of **asademo**.

ON clause This clause is supported from Interactive SQL only. In Interactive SQL, if *engine-name* is not specified, the default database is the first started server among those currently running.

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

Usage

Starts a specified database on the current database server.

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 632
- ♦ "CONNECT statement [ESQL] [Interactive SQL]" on page 332
- ♦ "-gd server option" [*ASA Database Administration Guide*, page 142]

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** Not applicable.

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]

Description

Use this statement to start a database server.

Syntax

START ENGINE AS *engine-name* [**STARTLINE** *command-string*]

Usage

The START ENGINE statement starts a database server. If you wish to specify a set of options for the server, use the STARTLINE keyword together with a command string. Valid command strings are those that conform to the database server description in "The database server" [*ASA Database Administration Guide,* page 116].

Permissions

None

Side effects

None

See also

- "STOP ENGINE statement" on page 633
- "The database server" [*ASA Database Administration Guide,* page 116]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

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 'dbeng9 -c 8M'
```

START JAVA statement

Description	Use this statement to start the Java VM.
Syntax	**START JAVA**
Usage	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.
Permissions	Java in the database 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 634
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not applicable.
Example	Start the Java VM.

```
START JAVA
```

START LOGGING statement [Interactive SQL]

Description	Use this statement to start logging executed SQL statements to a log file.
Syntax	**START LOGGING** *filename*
Usage	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 635
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not applicable.
Example	Start logging to a file called *filename.sql*, located in the c: directory.

```
START LOGGING 'c:\filename.sql'
```

START SUBSCRIPTION statement [SQL Remote]

Description	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.

Usage

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 400
- "REMOTE RESET statement [SQL Remote]" on page 575
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 639
- "sp_subscription procedure" [*SQL Remote User's Guide*, page 433]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

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]

Description

Use this statement to restart logging of deletes for MobiLink synchronization.

Syntax

START SYNCHRONIZATION DELETE

Usage

Ordinarily, Adaptive Server 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 will be 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 637
- “StartSynchronizationDelete method” [*UltraLite C/C++ User’s Guide*, page 317]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

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

Description

Use this statement to stop a database on the current database server.

Syntax

STOP DATABASE *database-name*
[**ON** *engine-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 *engine-name* is not specified in Interactive SQL, all running engines will be 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.

Usage

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 [Interactive SQL]" on page 623
- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 453
- "-gd server option" [*ASA Database Administration Guide*, page 142]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

Stop the database named *sample* on the current server.

```
STOP DATABASE sample
```

STOP ENGINE statement

Description

Use this statement to stop a database server.

Syntax

STOP ENGINE [*engine-name*] [**UNCONDITIONALLY**]

Parameters

STOP ENGINE clause The *engine-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.

Usage

The STOP ENGINE statement stops the specified database server. If the UNCONDITIONALLY keyword is supplied, the database server is stopped even if there are connections to the server. By default, the database server will not be stopped if there are connections to it.

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 625
- "-gk server option" [*ASA Database Administration Guide*, page 144]

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

Stop the current database server, as long as there are no other connections.

```
STOP ENGINE
```

STOP JAVA statement

Description	Use this statement to stop the Java VM.
Syntax	**STOP JAVA**
Usage	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	Java in the database must be installed and the database must be Java-enabled. This statement is not supported on Windows CE.
Side effects	None
See also	♦ "START JAVA statement" on page 626
Standards and compatibility	♦ **SQL/92** Vendor extension. ♦ **SQL/99** Vendor extension. ♦ **Sybase** Not applicable.
Example	Stop the Java VM.

```
STOP JAVA
```

STOP LOGGING statement [Interactive SQL]

Description

Use this statement to stop logging of SQL statements in the current session.

Syntax

STOP LOGGING

Usage

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 627

Example

The following example stops the current logging session.

```
STOP LOGGING
```

STOP SUBSCRIPTION statement [SQL Remote]

Description

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.

Usage

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 465
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 639

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Example

The following statement starts the subscription of user **SamS** to the **pub_contact** publication.

```
STOP SUBSCRIPTION TO pub_contact
FOR SamS
```

STOP SYNCHRONIZATION DELETE statement [MobiLink]

Description

Use this statement to temporarily stop logging of deletes for MobiLink synchronization.

Syntax

STOP SYNCHRONIZATION DELETE

Usage

Ordinarily, Adaptive Server 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 changes to an Adaptive Server Anywhere or UltraLite remote database.

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the subsequent delete operations executed on that connection will be 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

- "StartSynchronizationDelete method" [*UltraLite C/C++ User's Guide*, page 317]
- "StopSynchronizationDelete method" [*UltraLite C/C++ User's Guide*, page 317]
- "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 630

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

☞ For an example, see “START SYNCHRONIZATION DELETE statement [MobiLink]” on page 630.

SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]

Description

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.

Usage

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 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 400
- "START SUBSCRIPTION statement [SQL Remote]" on page 628

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** 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]

Description	Use this statement to launch an executable file from within Interactive SQL.
Syntax	**SYSTEM** ′ [*path*] *filename* ′

Usage

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 filename in single quotation marks.

Permissions

None.

Side effects

None.

See also

- “CONNECT statement [ESQL] [Interactive SQL]” on page 332

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** Not applicable.

Example

The following statement launches the Notepad program, assuming that the Notepad executable is in your path.

```
SYSTEM ’notepad.exe’
```

TRIGGER EVENT statement

Description

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, in order to simulate a context for the event handler.

Usage

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 273
- “CREATE EVENT statement” on page 351
- “EVENT_PARAMETER function [System]” on page 150

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));
```

TRUNCATE TABLE statement

Description

Use this statement to delete all rows from a table, without deleting the table definition.

Syntax

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

Usage

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.

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.

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.

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.

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.

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.

In UltraLite embedded SQL applications, use DELETE instead of TRUNCATE TABLE to delete all the rows in a table. TRUNCATE TABLE marks the rows for deletion, but unless you synchronize it does not physically remove them from the UltraLite database.

See also

- “DELETE statement” on page 445
- “TRUNCATE_WITH_AUTO_COMMIT option [database]” [*ASA Database Administration Guide*, page 699]

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

Delete all rows from the **department** table.

```
TRUNCATE TABLE department
```

UNION operation

Description

Use this statement to combine the results of two or more select statements.

Syntax

select-statement
  UNION [**ALL** | **DISTINCT**] *select-statement*
[**UNION** [**ALL** | **DISTINCT**] *select-statement*] . . .
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], . . .]

Usage

The results of several SELECT statements can be combined into a larger result using UNION. The component SELECT statements must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

The results of UNION ALL are the combined results of the component SELECT statements. 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, Adaptive Server Anywhere will choose a data type for the corresponding column in the result and automatically convert the columns in each component SELECT statement 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 SELECT statement. An alternative way of customizing result set column names is to use the WITH clause on the SELECT statement.

Permissions

Must have SELECT permission for each of the component SELECT statements.

Side effects

None.

See also

♦ “SELECT statement” on page 597

Standards and compatibility

♦ **SQL/92** Entry-level.

♦ **SQL/99** Core feature.

♦ **Sybase** Supported by Adaptive Server Enterprise, which also supports a COMPUTE clause.

Example

List all distinct surnames of employees and customers.

```
SELECT emp_lname
FROM Employee
UNION
SELECT lname
FROM Customer
```

UNLOAD statement

Description

Use this statement to export data from a database into an external ASCII-format file.

Syntax

UNLOAD *select-statement*
TO *filename*
[*unload-option* . . .]

unload-option :
  APPEND {**ON**|**OFF**}
| **DELIMITED BY** *string*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** {**ON** | **OFF**}
| **FORMAT** {**ASCII** | **BCP**}
| **HEXADECIMAL** {**ON** | **OFF**}
| **QUOTES** {**ON** | **OFF**}

filename : { *string* | *variable* }

Parameters

filename The filename to which the data is to be unloaded. Because it is the database server that executes the statements, filenames specify files on the database server machine. Relative filenames specify files relative to the database server's starting directory. To unload data onto a client machine, see "OUTPUT statement [Interactive SQL]" on page 556.

Usage

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 \x*nnnn* where *n* is a hexadecimal digit.

☞ For a description of the *unload-option* parameters, see "UNLOAD TABLE statement" on page 648.

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" [*ASA SQL User's Guide*, page 612].

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.

☞ For more information, see "-gl server option" [*ASA Database Administration Guide*, page 144].

Side effects

None. The query is executed at the current isolation level.

See also

- ♦ "UNLOAD TABLE statement" on page 648
- ♦ "OUTPUT statement [Interactive SQL]" on page 556

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** UNLOAD is not supported by Adaptive Server Enterprise.

UNLOAD TABLE statement

Description

Use this statement to export data from a database table into an external ASCII-format file.

Syntax

UNLOAD [**FROM**] **TABLE** [*owner.*]*table-name* **TO** *filename*
[*unload-option* . . .]

unload-option :
  APPEND{**ON** | **OFF**}
| **DELIMITED BY** *string*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** {**ON** | **OFF**}
| **FORMAT** {**ASCII** | **BCP**}
| **HEXADECIMAL** {**ON** | **OFF**}
| **ORDER** {**ON** | **OFF**}
| **QUOTES** {**ON** | **OFF**}

filename : { *string* | *variable* }

Parameters

filename The filename to which the data is to be unloaded. Because it is the database server that executes the statements, filenames specify files on the database server machine. Relative filenames specify files relative to the database server's starting directory. To unload data onto a client machine, see "OUTPUT statement [Interactive SQL]" on page 556.

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.

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 multi-byte 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" [*ASA SQL User's Guide*, page 63].

QUOTES option With QUOTES turned on (the default), single quotes are placed around all exported strings.

Usage

The UNLOAD TABLE statement allows efficient mass exporting from a database table into an ASCII file. UNLOAD TABLE 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, DELIMITED BY, and ESCAPE CHARACTER options, see "LOAD TABLE statement" on page 538.

Permissions

The permissions required to execute an UNLOAD TABLE statement depend on the database server `-gl` command line 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.

☞ For more information, see "-gl server option" [*ASA Database Administration Guide*, page 144].

Side effects

None.

See also

- ♦ "LOAD TABLE statement" on page 538
- ♦ "OUTPUT statement [Interactive SQL]" on page 556
- ♦ "UNLOAD statement" on page 646

Standards and compatibility

- ♦ **SQL/92** Vendor extension.
- ♦ **SQL/99** Vendor extension.
- ♦ **Sybase** UNLOAD TABLE is not supported by Adaptive Server Enterprise. Similar functionality is provided by the Adaptive Server Enterprise bulk copy utility (**bcp**).

UPDATE statement

Description

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**], . . .]

Syntax 2

UPDATE *table-name*
SET *set-item*, . . .
VERIFY (*column-name*, . . . **) VALUES (** *expression*, . . . **)**
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], . . .]

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*
| *column-name*[.*field-name*. . .].*method-name***(** [*expression*] **)**
| @*variable-name* **=** *expression*

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 operation.

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 If you are updating Java columns, you can use *field-name* to update the value of a public field in the column. Alternatively, you can use a method to set the value. The following clause updates name field of the *JProd* column using a method:

```
SET JProd.setName( ’Tank Top’ )
```

If you are updating non-Java columns, the SET clause is of the following form:

```
SET column-name = expression, ...
```

and/or

```
SET @variable-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. 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]” [*ASA Database Administration Guide*, page 635].

☞ For a full description of joins, see “Joins: Retrieving Data from Several Tables” [*ASA SQL User’s Guide*, page 263].

☞ For more information, see “FROM clause” on page 491.

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]" [*ASA Database Administration Guide*, page 635].

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 upper case V and the remainder of the letters lower case. 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.

Usage

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 in order 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 will also update column statistics.

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

Column statistics are updated.

See also

- "DELETE statement" on page 445
- "INSERT statement" on page 528
- "FROM clause" on page 491
- "Joins: Retrieving Data from Several Tables" [*ASA SQL User's Guide,* page 263]

Standards and compatibility

- **SQL/92** Syntax 1 is an entry-level 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.
- **SQL/99** 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/92 compatibility, ensure that the ANSI_UPDATE_CONSTRAINTS option is set to STRICT.

  ☞ For more information, see "ANSI_UPDATE_CONSTRAINTS option [compatibility]" [*ASA Database Administration Guide,* page 635].
- **Sybase** Subject to the expressions being compatible, the syntax of the UPDATE statement (syntax 1) is compatible between Adaptive Server Enterprise and Adaptive Server Anywhere. Syntax 2 and 3 are not supported.

Example

Transfer employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 129;
```

Sales orders currently start at ID 2001. Renumber all existing sales orders by subtracting 2000 from the ID.

```
UPDATE sales_order AS orders
SET orders.id = orders.id - 2000
ORDER BY items.id ASC
```

This update is possible only if the foreign key of the sales_order_items table (referencing the primary key sales_order.id) is defined with the action ON UPDATE CASCADE. The sales_order_items table is then updated as well.

☞ For more information on foreign key properties, see "ALTER TABLE statement" on page 294 and "CREATE TABLE statement" on page 407.

UPDATE (positioned) statement [ESQL] [SP]

Description

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 *table-list*
SET *set-item*, . . .
WHERE CURRENT OF *cursor-name*

hostvar-list : *indicator variables allowed*

set-item :
  column-name [.*field-name*. . .] = *expression*
| *column-name* [.*field-name*. . .].*method-name***(** [*expression*] **)**

sqlda-name : *identifier*

Parameters

SET clause The columns that are referenced in *set-item* must be in the base table that is updated. They cannot refer to aliases, nor to columns from other tables or views. If the table you are updating is given a correlation name in the cursor specification, you must use the correlation name in the SET 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.

The expression on the right side of the SET clause 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 cannot contain aggregate functions, subqueries, or subselects.

Usage

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.

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 445
- “DELETE (positioned) statement [ESQL] [SP]” on page 447
- “UPDATE statement” on page 650

Standards and compatibility

- **SQL/92** Entry-level feature. The range of cursors that can be updated may contain vendor extensions if the ANSI_UPDATE_CONSTRAINTS option is set to OFF.
- **SQL/99** 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 Adaptive Server Anywhere.

Example

The following is an example of an UPDATE statement WHERE CURRENT OF cursor:

```
UPDATE Employee
SET emp_lname = 'Jones'
WHERE CURRENT OF emp_cursor;
```

UPDATE statement [SQL Remote]

Description

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*

Usage

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 will be used.

If no WHERE clause is specified, every row will be updated. If a WHERE clause is specified, then only those rows which satisfy the search condition will be updated.

Normally, the order that rows are updated doesn't 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 wish 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 operation.

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 lower case. 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 in order to ensure that the rows are kept in their proper subscriptions.

☞ For a full description of this feature, see "Territory realignment in the Contact example" [*SQL Remote User's Guide,* page 107].

☞ 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.

See also

- "CREATE TRIGGER statement [SQL Remote]" on page 423

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.

Examples

- Transfer employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE employee
VERIFY( dept_id ) VALUES( 300 )
SET dept_id = 400
WHERE emp_id = 129
```

VALIDATE CHECKSUM statement

Description

Use this statement to validate a database using checksums.

Syntax

VALIDATE CHECKSUM

Usage

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.

This type of validation can only be used on databases that were created with checksums enabled. You can check whether a database has checksums enabled by executing the following statement:

```
SELECT db_property ( 'Checksum' )
```

Permissions

None

Side effects

None

See also

- "CREATE DATABASE statement" on page 338
- "sa_validate system procedure" on page 838
- "The Validation utility" [*ASA Database Administration Guide*, page 604]

VALIDATE INDEX statement

Description

Use this statement to validate an index, including index statistics.

Syntax

VALIDATE INDEX [[*owner*.]*table-name*.] { *index-name* | *table-name* }

Usage

Ensures that every row referenced in the index actually exists in the table. 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 index(es) are accurate. If they are not accurate, an error is generated.

index-name | table-name If you supply a *table-name* instead of an *index-name*, the primary key index is validated.

Permissions

Must be the owner of the table on which the index is created, have DBA authority, or have REMOTE DBA authority (SQL Remote).

Side effects

None.

See also

- "CREATE INDEX statement" on page 368
- "VALIDATE TABLE statement" on page 662
- "The Validation utility" [*ASA Database Administration Guide*, page 604]

VALIDATE TABLE statement

Description	Use this statement to validate a table in the database.
Syntax	**VALIDATE TABLE** [*owner*.]*table-name* [**WITH** { **DATA** | **EXPRESS** | **FULL** | **INDEX** } **CHECK**]
Parameters	**WITH DATA CHECK** If you have LONG BINARY, LONG VARCHAR, TEXT, or IMAGE entries, they may span more than one database page. In addition to the default checks, this option instructs the database server to check all pages used by each entry. **WITH EXPRESS CHECK** In addition to the default and WITH DATA checks, check that the number of rows in the table matches the number of entries in the index. This option does not perform individual index lookups for each row. This option can significantly improve performance when validating large databases with a small cache. **WITH FULL CHECK** In addition to the default checks, carry out a DATA CHECK and an INDEX CHECK. **WITH INDEX CHECK** In addition to the default checks, validate each index on the table. For information on index validation, see "VALIDATE INDEX statement" on page 661.
Usage	With no additional options, VALIDATE TABLE scans every row of a table. For each entry that is in an index, it checks the validity of the database page that the entry starts on, and checks that an entry for the row exists in the proper index. The VALIDATE TABLE statement also ensures, for each index in the table, that the number of rows referenced by the index is not greater than the number of rows in the table. This default validation is sufficient for most purposes. Options are provided for additional validation, which may be helpful in unusual circumstances. Depending on the contents of your database, these additional checks may significantly extend the time required to validate. If the table is corrupt, an error is reported. If you do have errors reported, you can drop all of the indexes and keys on a table and recreate them. Any foreign keys to the table will also need to be recreated. Another solution to errors reported by VALIDATE TABLE is to unload and reload your entire database. You should use the -u option of DBUNLOAD so that it will not try to use a possibly corrupt index to order the data.
Permissions	Must be the owner of the table, have DBA authority, or have REMOTE DBA authority (SQL Remote).
Side effects	None.

See also

- "The Validation utility" [*ASA Database Administration Guide*, page 604]
- "VALIDATE INDEX statement" on page 661
- "sa_validate system procedure" on page 838

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** VALIDATE TABLE is not supported in Adaptive Server Enterprise. The procedure **dbcc checktable** provides a similar function.
- **WITH EXPRESS CHECK option** This option is only supported for databases created with Adaptive Server Anywhere version 7.0 or later.

WAITFOR statement

Description

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 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" [*ASA Database Administration Guide*, page 713].

Usage

The WAITFOR statement wakes up periodically (every 5 seconds by default) to check if it has been cancelled 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, and may be useful for customers who choose not to license Java in the database:

```
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 351

Standards and compatibility

- **SQL/92** Vendor extension.
- **SQL/99** Vendor extension.
- **Sybase** This statement is also implemented by Adaptive Server Enterprise.

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]

Description

Use this statement to specify error handling in embedded SQL programs.

Syntax

WHENEVER { **SQLERROR** | **SQLWARNING** | **NOTFOUND** }
GOTO *label* | **STOP** | **CONTINUE** | { *C-code*; }

label : *identifier*

Usage

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/92** Entry-level feature.
- **SQL/99** Core feature.
- **Sybase** Supported by Open Client/Open Server.

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]

Description

Use this statement to provide repeated execution of a statement or compound statement.

Syntax

WHILE *search-condition-statement*

Usage

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 548

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

Example

The following code illustrates the use of WHILE:

```
WHILE ( SELECT AVG(unit_price) FROM product ) < $30
BEGIN
   UPDATE product
   SET unit_price = unit_price + 2
   IF ( SELECT MAX(unit_price) FROM product ) > $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.

WRITETEXT statement [T-SQL]

Description

Permits non-logged, interactive updating of an existing text or image column.

Syntax

WRITETEXT *table-name*.*column-name*
text_pointer [**WITH LOG**] *data*

Usage

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 573
- "TEXTPTR function [Text and image]" on page 234

Standards and compatibility

- **SQL/92** Transact-SQL extension.
- **SQL/99** Transact-SQL extension.
- **Sybase** Supported by Adaptive Server Enterprise.

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 part describes system tables, views, and procedures.

CHAPTER 5

System Tables

About this chapter

The structure of every database is described in a number of system tables.

The system tables are owned by the **SYS** user ID. 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.

This chapter contains descriptions of each of the system tables. Several of the columns have only two possible values. Usually these values are **Y** and **N** for **yes** and **no** respectively. These columns are designated by **(Y/N)**.

Contents

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. Further, the SYS.DUMMY table does not appear in a Lock node in the graphical plan, and if only the SYS.DUMMY table is present, a Lock node is not added.

RowGenerator system table

Column name	Column type	Column constraint	Table constraints
row_num	SMALLINT	NOT NULL	

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 stored procedures and views, and should not be modified in any way.

You can also use the sa_rowgenerator stored procedure to generate a range of numeric values.

☞ For more information, see “sa_rowgenerator system procedure” on page 827.

row_num A value between 1 and 255.

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 dbo.RowGenerator
where row_num < 32
and datepart(month,day_of_month) =
   datepart(month,current date)
order by row_num
```

SYSARTICLE system table

Column name	Column type	Column constraint	Table constraints
publication_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSPUBLICATION.
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSTABLE.
where_expr	LONG VARCHAR		
subscribe_by_expr	LONG VARCHAR		
query	CHAR(1)	NOT NULL	

Each row of SYSARTICLE describes an article in a publication.

publication_id The publication of which this 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.

SYSARTICLECOL system table

Column name	Column type	Column constraint	Table constraints
publication_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSARTICLE.
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSARTICLE, SYSCOLUMN.
column_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.

Each row identifies a column in an article.

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 SYSCOLUMN system table.

SYSATTRIBUTE system table

SYSATTRIBUTE and SYSATTRIBUTENAME were created so that new information about database objects could be added to the system tables without changing the schema.

Each row of SYSATTRIBUTE describes one system object, such as a particular table or index. Rows are added when the attribute is specified; for example, every table with a table page percent free setting is added to SYSATTRIBUTE. Similarly, a row for index depth appears in SYSATTRIBUTE only when the index depth increases to 2.

Statistics are maintained for all indexes, including those on catalog tables, as each index is updated. The VALIDATE statement verifies that the statistics on the specified index(es) are accurate and generates an error if they are not. This provides accurate statistics to the optimizer at virtually no performance cost. Statistics persist in SYSATTRIBUTE in the form of one row for each statistic for an index.

	object_type	**object_id**	**attribute_-id (from SYSAT-TRIBUTE-NAME**	**sub_-object_id1**	**sub_-object_id2**	**attribute_-value**
Table page percent free	T	table_id		NULL	NULL	percentage of free space left in each table page
Clustered index	T	table_id	2	index_-number	NULL	NULL
Number of distinct key values	I	table_id	3	NULL	NULL	number of distinct key values
Number of leaf pages	I	table_id	4	index_-number	NULL	number of leaf pages
Index depth	I	table_id	5	index_-number	NULL	index depth

	object_type	object_id	attribute_-id (from SYSATTRIBUTE-NAME	sub_-object_id1	sub_-object_id2	attribute_-value
Maximum index key value distance	I	table_id	6	index_-number	NULL	maximum distance, in pages, between rows containing the same key value

Columns in the SYSATTRIBUTE table exhibit the following characteristics:

Column name	Column type	Column constraint
object_type	CHAR(1)	NOT NULL
object_id	UNSIGNED INT	NOT NULL
attribute_id	UNSIGNED INT	NOT NULL. Foreign key references SYSATTRIBUTE-NAME.
sub_object_id1	UNSIGNED INT	
sub_object_id2	UNSIGNED INT	
attribute_value	LONG VARCHAR	

object_type The type of object the attribute describes. For example, T represents a table, I an index.

object_id The id of the particular object. Object id information can be obtained from the table_id column after executing `SELECT * FROM SYSTABLE`.

attribute_id The number representing the attribute that is being described. A descriptive name for each attribute ID is stored in SYSATTRIBUTENAME.

sub_object_id1 Additional information about the attribute, or NULL if there is none.

sub_object_id2 Additional information about the attribute, or NULL if there is none.

attribute_value	The value of the attribute.

SYSATTRIBUTENAME system table

Column name	Column type	Column constraint
attribute_id	UNSIGNED INT	NOT NULL.
attribute_name	CHAR(128)	NOT NULL.

This table provides attribute names for the attribute IDs that are used in SYSATTRIBUTE.

attribute_id The ID of the attribute.

attribute_name The name of the attribute.

SYSCAPABILITY system table

Column name	Column type	Column constraint	Table constraints
capid	INTEGER	NOT NULL	Primary key. Foreign key references SYSCAPABILITYNAME.
srvid	INTEGER	NOT NULL	Primary key. Foreign key references SYSSERVERS.
capvalue	CHAR(128)	NOT NULL	

Each row identifies a capability of a remote server.

capid The capability, as listed in SYSCAPABILITYNAME.

srvid The server to which the capability applies, as listed in SYSSERVERS.

capvalue The value of the capability.

SYSCAPABILITYNAME system table

Column name	Column type	Column constraint	Table constraints
capid	INTEGER	NOT NULL	Primary key.
capname	CHAR(128)	NOT NULL	

Each row identifies a capability.

capid The capability ID.

capname The name of the capability.

SYSCHECK system table

Column name	Column type	Column constraint	Table constraints
check_id	INTEGER	NOT NULL	Primary key. Foreign key references SYSCONSTRAINT.
check_defn	LONG VARCHAR	NOT NULL	

Each row identifies a named check constraint in a table.

check_id An identifier for the constraint.

check_defn The CHECK expression.

SYSCOLLATION system table

Column name	Column type	Column constraint	Table constraint
collation_id	SMALLINT	NOT NULL	Primary key.
collation_label	CHAR(10)	NOT NULL	
collation_name	CHAR(128)	NOT NULL	
collation_order	BINARY(1280)	NOT NULL	

This table contains the collation sequence information for the database. There is no way to modify the contents of this table.

collation_id A unique number identifying the collation sequence.

collation_label A string identifying the collation sequence. The collation sequence to be used is selected when the database is created, by specifying the collation label with the `-z` option.

collation_name The name of the collation sequence.

collation_order An array of bytes defining how each of the 256 character codes are treated for comparison purposes. All string comparisons translate each character according to the collation order table before comparing the characters. For the different ASCII code pages, the only difference is how accented characters are sorted. In general, an accented character is sorted as if it were the same as the nonaccented character.

SYSCOLLATIONMAPPINGS system table

Column name	Column type	Column constraint	Table Constraints
collation_label	CHAR(10)	NOT NULL	Primary key.
collation_name	CHAR(128)	NOT NULL	
cs_label	CHAR(128)		
so_case_label	CHAR(128)		
so_caseless_label	CHAR(128)		
jdk_label	CHAR(128)		

collation_label A string identifying the collation sequence. The collation sequence to be used is selected when the database is created, by specifying the collation label with the `-z` option.

collation_name The collation name used to describe the character set encoding.

cs_label The Adaptive Server Enterprise character set mapping label.

so_case_label The collation sort order for case-sensitive Adaptive Server Enterprise character set mapping.

so_caseless_label The collation sort order for case-insensitive Adaptive Server Enterprise character set mapping.

jdk_label The JDK character set label.

This table contains only one row with the database collation mapping.

SYSCOLPERM system table

Column name	Column type	Column constraint	Table constraint
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.
grantee	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.
grantor	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.
column_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.
privilege_type	SMALLINT	NOT NULL	Primary key.
is_grantable	CHAR(1)	NOT NULL	

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 SYSCOLPERM.

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.

SYSCOLSTAT system table

Column name	Column type	Column constraint	Table constraints
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.
column_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.
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		

This table stores the column statistics that are stored as histograms and used by the optimizer. The contents of this table are best retrieved using the sa_get_histogram stored procedure or the Histogram utility.

table_id A number that uniquely identifies the table or view to which this column belongs.

column_id A number that, together with table_id, uniquely identifies the column.

format_id Internal field used to determine the format of the rest of the row.

update_time The time of the last update of this row.

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.

SYSCOLUMN system table

Column name	**Column type**	**Column constraint**	**Table constraints**
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSTABLE.
column_id	UNSIGNED INT	NOT NULL	Primary key.
pkey	CHAR(1)	NOT NULL	
domain_id	SMALLINT	NOT NULL	Foreign key references SYSDOMAIN.
nulls	CHAR(1)	NOT NULL	
width	SMALLINT	NOT NULL	
scale	SMALLINT	NOT NULL	
unused	INTEGER	NOT NULL	
max_identity	BIGINT	NOT NULL	
column_-name	CHAR(128)	NOT NULL	
remarks	LONG VAR-CHAR		
”default”	LONG VAR-CHAR		
unused2	LONG VAR-CHAR		
user_type	SMALLINT		Foreign key references SYSUSERTYPE.
format_str	CHAR(128)		
column_type	CHAR(1)	NOT NULL	
remote_name	VAR-CHAR(128)		
remote_type	UNSIGNED INT		

Each column in every table or view is described by one row in SYSCOLUMN.

table_id A number that uniquely identifies the table or view to which this column belongs.

column_id Each table starts numbering columns at 1. The order of column numbers determines the order that columns are displayed in the command

```
SELECT * FROM TABLE
```

pkey (Y/N) Indicate whether this column is part of the primary key for the table.

domain_id The data type for the column, indicated by a data type number listed in the SYSDOMAIN table.

nulls (Y/N) Indicates whether the NULL value is allowed in this 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 data type columns, and zero for all other data types.

unused Not used.

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.

remarks A comment string.

default The default value for the column. This value is only used when an INSERT statement does not specify a value for the column.

unused2 Not used.

user_type If the column is defined using a user-defined data type, the data type is held here.

format_str Currently unused.

column_type The type of column. Contains C for a computed column and R for other columns.

remote_name For columns in proxy tables, the name of the remote column.

remote_type For columns in proxy tables, the type of the remote column. This value is defined by the remote server or interface.

SYSCONSTRAINT system table

Column name	**Column type**	**Column constraint**	**Table constraints**
constraint_id	INTEGER	NOT NULL	Primary key.
table_id	INTEGER	NOT NULL	Foreign key references SYSTABLE, SYSCOLUMN, and SYSINDEX.
column_id	INTEGER		Foreign key references SYSCOLUMN.
index_id	INTEGER		Foreign key references SYSINDEX.
fkey_id	SMALLINT		Foreign key references SYSFOREIGNKEY.
constraint_type	CHAR(1)	NOT NULL	
constraint_name	CHAR(128)	NOT NULL	

Each row describes a named constraint. This table does not contain rows for catalog tables.

constraint_id The unique constraint ID.

table_id The table ID of the table to which the constraint applies.

column_id The column ID of the column to which the constraint applies. The column is NULL for any constraints that are not column constraints.

index_id The index ID for a unique constraint. The column is NULL for all constraints that are not unique constraints.

fkey_id The foreign key ID for a foreign key constraint. The column is NULL for all constraints that are not foreign key constraints.

constraint_type Set to one of the following values:

- ♦ C if the constraint is a column check constraint.
- ♦ T if the constraint is a table constraint.

- P if the constraint is a primary key.
- F if the constraint is a foreign key.
- U if the constraint is a unique constraint.

constraint_name	The name of the constraint.

SYSDOMAIN system table

Column name	Column type	Column constraint	Table constraints
domain_id	SMALLINT	NOT NULL	Primary key.
domain_name	CHAR(128)	NOT NULL	
type_id	SMALLINT	NOT NULL	
precision	SMALLINT		

Each of the predefined data types (sometimes called **domains**) is assigned a unique number. The SYSDOMAIN table is provided for informational purposes, to show the association between these numbers and the appropriate data types. This table is never changed.

domain_id The unique number assigned to each data type. These numbers cannot be changed.

domain_name A string containing the data type normally found in the CREATE TABLE command, such as **char** or **integer**.

type_id The ODBC data type. This corresponds to “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.

SYSEVENT system table

Column name	Column type	Column constraint	Table constraints
event_id	INTEGER	NOT NULL	Primary key.
creator	UNSIGNED INT	NOT NULL	Foreign key references SYSUSERPERM.
event_name	VARCHAR(128)	NOT NULL	Foreign key references SYSEVENTTYPE.
enabled	CHAR(1)	NOT NULL	
location	CHAR(1)	NOT NULL	
event_type_id	INTEGER		
action	LONG VARCHAR		
external_action	LONG VARCHAR		
condition	LONG VARCHAR		
remarks	LONG VARCHAR		
source	LONG VARCHAR		

Each row in SYSEVENT describes an event created with CREATE EVENT.

event_id The unique number assigned to each event.

creator The user number of the owner of the event. The name of the user can be found by looking in SYSUSERPERM.

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 SYSEVENTTYPE.

action The event handler definition.

external_action Not used.

condition The WHERE condition used to control firing of the event handler.

remarks A comment string.

source This column contains the original source for the event handler if the preserve_source_format option is ON. It is used to maintain the appearance of the original text. For more information, see "PRESERVE_SOURCE_FORMAT option [database]" [*ASA Database Administration Guide,* page 683].

SYSEVENTTYPE system table

Column name	Column type	Column constraint	Table constraints
event_type_id	INTEGER	NOT NULL	Primary key.
name	VARCHAR(128)	NOT NULL	
description	LONG VARCHAR		

This table lists the system event types which can be referenced by CREATE EVENT.

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.

SYSEXTENT system table

Column name	Column type	Column constraint	Table constraint
file_id	SMALLINT	NOT NULL	Primary key. Foreign key references SYSFILE.
extent_id	SMALLINT	NOT NULL	Primary key.
first_page	INTEGER	NOT NULL	
last_page	INTEGER	NOT NULL	
file_name	LONG VARCHAR	NOT NULL	

This table is not used.

SYSEXTERNLOGINS system table

Column name	Column type	Column constraint	Table constraints
user_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key to SYSUSERPERM.
srvid	INTEGER	NOT NULL	Primary key. Foreign key to SYSSERVERS.
remote_login	VARCHAR(128)		
remote_password	VARBINARY(128)		

Each row describes an external login for remote data access.

user_id The user ID on the local database.

srvid The remote server, as listed in SYSSERVERS.

remote_login The login name for this user, for the remote server.

remote_password The password for this user, for the remote server.

SYSFILE system table

Column name	Column type	Column constraint	Table constraint
file_id	SMALLINT	NOT NULL	Primary key.
file_name	LONG VARCHAR	NOT NULL	Unique index.
dbspace_name	CHAR(128)	NOT NULL	
store_type	CHAR(8)	NOT NULL	

Every database consists of one or more dbspaces. Each dbspace corresponds to an operating system file. Each dbspace is recorded in SYSFILE.

file_id Each file in a database is assigned a unique number. This file identifier is the primary key for SYSFILE. All system tables are stored in **file_id** 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. This value is for informational purposes only and 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 Every file has a dbspace name that is unique. It is used in the CREATE TABLE command.

store_type This field is for internal use.

SYSFKCOL system table

Column name	Column type	Column constraint	Table constraints
foreign_table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN and SYSFOREIGNKEY.
foreign_key_id	SMALLINT	NOT NULL	Primary key. Foreign key references SYSFOREIGNKEY.
foreign_column_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN.
primary_column_id	UNSIGNED INT	NOT NULL	

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.

foreign_table_id The table number of the foreign table.

foreign_key_id The key number of the FOREIGN KEY for the foreign table. Together, foreign_table_id and foreign_key_id uniquely identify one row in SYSFOREIGNKEY. The table number for the primary table can be obtained from that row (using the SYSFOREIGNKEY table).

foreign_column_id This column number and the foreign_table_id identify the foreign column description in SYSCOLUMN.

primary_column_id This column number and the primary_table_id obtained from SYSFOREIGNKEY identify the primary column description in SYSCOLUMN.

SYSFOREIGNKEY system table

Column name	Column type	Column constraint	Table constraints
foreign_table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSTABLE.table_id. Unique index.
foreign_key_id	SMALLINT	NOT NULL	Primary key.
primary_table_id	UNSIGNED INT	NOT NULL	Foreign key references SYSTABLE.table_id.
root	INTEGER	NOT NULL	
check_on_commit	CHAR(1)	NOT NULL	
nulls	CHAR(1)	NOT NULL	
role	CHAR(128)	NOT NULL	Unique index.
remarks	LONG VARCHAR		
primary_index_id	UNSIGNED INT	NOT NULL	
fk_not_enforced	CHAR(1)	NOT NULL	
hash_limit	SMALLINT	NOT NULL	

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.

foreign_table_id The table number of the foreign table.

foreign_key_id Each foreign key has a foreign key number that is unique with respect to:

- The key number of all other foreign keys for the foreign table

- The key number of all foreign keys for the primary table
- The index number of all indexes for the foreign table

primary_table_id The table number of the primary table.

root The **root** identifies the location of the root page of the foreign key in the database file.

check_on_commit (Y/N) Indicates whether INSERT and UPDATE statements should wait until the next COMMIT to check if foreign keys are valid. A foreign key is valid if, for each row in the foreign table, the values in the columns of the foreign key either contain the NULL value or match the primary key values in some row of the primary table.

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.

role The name of the relationship between the foreign table and the primary table. Unless otherwise specified, the **role** name will be the same as the name of the primary table. The foreign table cannot have two foreign keys with the same role name.

remarks A comment string.

primary_index_id The index_id of the primary key.

fk_not_enforced (Y/N) Is N if one of the tables is remote.

hash_limit Contains information about physical index representation.

SYSGROUP system table

Column name	**Column type**	**Column constraint**	**Table constraints**
group_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.user_id.
group_member	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.user_id.

There is one row in SYSGROUP for every member of every group. This table describes a many-to-many relationship between groups and members. A group may have many members, and a user may be a member of many groups.

group_id The user number of the group.

group_member The user number of a member.

SYSHISTORY system table

Column name	Column type	Column constraint	Table constraints
operation	CHAR(12)	NOT NULL	Primary key.
version	CHAR(15)	NOT NULL	Primary key.
platform	CHAR(50)	NOT NULL	Primary key.
first_time	TIMESTAMP	NOT NULL	
last_time	TIMESTAMP	NOT NULL	
details	VARCHAR(2048)		

A new row is added to the SYSHISTORY table each time the database is started with a different version of the software and/or on a different platform. Each software version/platform combination is entered only once in the table.

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.

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.

SYSINDEX system table

Column name	Column type	Column constraint	Table constraints
table_id	UNSIGNED INT	NOT NULL	Primary key, unique index. Foreign key references SYSTABLE.
index_id	UNSIGNED INT	NOT NULL	Primary key.
root	INTEGER	NOT NULL	
file_id	SMALLINT	NOT NULL	
”unique”	CHAR(1)	NOT NULL	
creator	UNSIGNED INT	NOT NULL	Foreign key references SYSUSERPERM.user_id.
index_name	CHAR(128)	NOT NULL	Unique index.
hash_limit	SMALLINT	NOT NULL	
index_owner	CHAR(4)	NOT NULL	
index_type	CHAR(4)	NOT NULL	
remarks	LONG VARCHAR		

Each index in the database is described by one row in SYSINDEX. Each column in the index is described by one row in SYSIXCOL.

table_id Uniquely identifies the table to which this index applies.

index_id Each index for one particular table is assigned a unique index number.

root The **root** identifies the location of the root page of the index in the database file.

file_id The index is completely contained in the file with this **file_id** (see SYSFILE).

unique Indicates whether the index is a unique index (“Y”), a non-unique index (“N”), or a unique constraint (“U”). A unique index prevents two rows in the indexed table from having the same values in the index columns.

creator The user number of the creator of the index. This user is always the same as the creator of the table identified by table_id.

index_name The name of the index. A table cannot have two indexes with the same name.

index_owner The owner. This field is always **USER**.

index_type The type. This field is always SA.

remarks A comment string.

SYSINFO system table (deprecated)

Column name	Column type	Column constraint	Table constraints
page_size	INTEGER	NOT NULL	
encryption	CHAR(1)	NOT NULL	
blank_padding	CHAR(1)	NOT NULL	
case_sensitivity	CHAR(1)	NOT NULL	
default_collation	CHAR(10)		
database_version	SMALLINT	NOT NULL	
classes_version	CHAR(10)		

This table indicates the database characteristics, as defined when the database was created. It always contains only one row.

Note
Use of this table is deprecated. It is recommended that you use the db_property function instead. The following parameter values to db_property can be used to obtain The information in SYSINFO:

- ‘PageSize’
- ‘Encryption’
- ‘BlankPadding’
- ‘CaseSensitive’
- ‘Collation’
- ‘FileVersion’
- ‘JDKVersion’

☞ For more information about The db_property function, see “DB_PROPERTY function [System]” on page 139.

page_size The page size specified, in bytes. The default value is 1024.

encryption (Y/N) Indicates whether The database is encrypted.

blank_padding (Y/N) Indicates whether The database was created to use blank padding for string comparisons in The database.

case_sensitivity (Y/N) Indicates whether The database is created as case sensitive. Case sensitivity affects value comparisons, but not table and column name comparisons. For example, if a database is case sensitive, table names such as SYSCATALOG can be specified in either case, but in a case-sensitive database `'abc' = 'ABC'` is not true.

default_collation A string corresponding to The collation_label in SYSCOLLATION, which also corresponds to the collation sequence specified with the Initialization utility. The collation sequence is used for all string comparisons, including searches for character strings as well as column and table name comparison.

database_version A small integer value indicating the database format.

classes_version A small string describing the current version of the Java runtime library that is used by this database.

SYSIXCOL system table

Column name	Column type	Column constraint	Table constraints
table_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSCOLUMN. Foreign key references SYSINDEX.
index_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSINDEX.
sequence	SMALLINT	NOT NULL	Primary key.
column_id	UNSIGNED INT	NOT NULL	Foreign key references SYSCOLUMN.
”order”	CHAR(1)	NOT NULL	

Every index has one row in SYSIXCOL for each column in the index.

table_id Identifies the table to which the index applies.

index_id Identifies in which index this column is used. Together, table_id and index_id identify one index described in SYSINDEX.

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 The column number identifies which column is indexed. Together, table_id and column_id identify one column described in SYSCOLUMN.

order (A/D) Indicates whether this column in the index is kept in ascending or descending order.

SYSJAR system table

Column name	Column type	Column constraint	Table constraints
jar_id	INTEGER	NOT NULL	Primary key.
creator	UNSIGNED INT	NOT NULL	
jar_name	LONG VARCHAR	NOT NULL	Unique index.
jar_file	LONG VARCHAR		
create_time	TIMESTAMP	NOT NULL	
update_time	TIMESTAMP	NOT NULL	
remarks	LONG VARCHAR		

jar_id A field containing the id of the JAR file.

creator The user number of the creator of the JAR file.

jar_name The name of the JAR file.

jar_file The file name of the JAR file.

create_time The time the JAR file was created.

update_time The time the JAR file was last updated.

remarks A comment field.

SYSJARCOMPONENT system table

Column name	Column type	Column constraint	Table constraints
component_id	INTEGER	NOT NULL	Primary key.
jar_id	INTEGER		Foreign key references SYSJAR.
component_name	LONG VARCHAR		
component_type	CHAR(1)		
create_time	TIMESTAMP	NOT NULL	
contents	LONG BINARY		
remarks	LONG VARCHAR		

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.

create_time A field containing the creation time of the component.

contents The byte code of the JAR file.

remarks A comment field.

SYSJAVACLASS system table

Column name	Column type	Column constraint	Table constraints
class_id	INTEGER	NOT NULL	Primary key.
replaced_by	INTEGER		Foreign key references SYSJAVACLASSES.class_id.
creator	UNSIGNED INT	NOT NULL	Foreign key references SYSUSERPERM.user_id.
jar_id	INTEGER		
type_id	SMALLINT		Foreign key references SYSUSERTYPE.
class_name	LONG VARCHAR	NOT NULL	
public	CHAR(1)	NOT NULL	
component_id	INTEGER		Foreign key references SYSJARCOMPONENT.
create_time	TIMESTAMP	NOT NULL	
update_time	TIMESTAMP	NOT NULL	
class_descriptor	LONG BINARY		
remarks	LONG VARCHAR		

The SYSJAVACLASS system table contains all information related to Java classes.

class_id This field contains the id of the java class. Also the primary key for the table.

replaced_by A field that references the primary key field, class_id.

creator The user number of the creator of the class.

jar_id The id of the JAR file from which the class came.

type_id The id of the user type.

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 SYSJARCOMPONENT.

create_time Contains the creation time of the class.

update_time Contains the last update time of the class.

class_descriptor Not used.

remarks Contains a comment string.

SYSLOGIN system table

Column name	Column type	Column constraint	Table constraints
integrated_login_id	CHAR(128)	NOT NULL	Primary key.
login_uid	UNSIGNED INT	NOT NULL	Foreign key references SYSUSERPERM.user_id.
remarks	LONG VARCHAR		

This table contains all the User Profile names that can be used to connect to the database using an integrated logon. As a security measure, only users with DBA authority can view the contents of this table.

integrated_login_id A string value containing the User Profile name that is used to map to a user ID in the database. When a user successfully logs on using this User Profile name, and the database is enabled to accept integrated logons, the user can connect to the database without providing a user ID or password.

login_uid A foreign key to the system table SYSUSERPERM.

remarks A comment string

SYSOPTBLOCK system table

This table is reserved for system use.

SYSOPTION system table

Column name	Column type	Column constraint	Table constraints
user_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSER-PERM.
”option”	CHAR(128)	NOT NULL	Primary key.
”setting”	LONG VAR-CHAR	NOT NULL	

Options settings are stored in the SYSOPTION table by the SET command. Each user can have their own setting for each option. In addition, settings for the PUBLIC user ID define the default settings to be used for user IDs that do not have their own setting.

user_id The user number to whom this option setting applies.

option The name of the option.

setting The current setting for the named option.

SYSOPTJOINSTRATEGY system table

This table is reserved for system use.

SYSOPTORDER system table

This table is reserved for system use.

SYSOPTQUANTIFIER system table

This table is reserved for system use.

SYSOPTREQUEST system table

This table is reserved for system use.

SYSOPTREWRITE system table

This table is reserved for system use.

SYSOPTSTAT system table

This table stores information about the cost model. It is reserved for system use.

SYSPROCEDURE system table

Column name	Column type	Column constraint	Table constraints
proc_id	UNSIGNED INT	NOT NULL	Primary key.
creator	UNSIGNED INT	NOT NULL	Unique index on this column and proc_name. Foreign key references SYSUSERPERM.-user_id.
proc_name	CHAR(128)	NOT NULL	Unique index on this column and creator.
proc_defn	LONG VARCHAR		
remarks	LONG VARCHAR		
replicate	CHAR(1)	NOT NULL	
srvid	INTEGER		Foreign key references SYSSERVERS.
source	LONG VARCHAR		
avg_num_rows	FLOAT		
avg_cost	FLOAT		
stats	LONG BINARY		

Each procedure in the database is described by one row in SYSPROCEDURE.

proc_id Each procedure is assigned a unique number (the **procedure number**), which is the primary key for SYSPROCEDURE.

creator This user number identifies the owner of the procedure. The name of the user can be found by looking in SYSUSERPERM.

proc_name The name of the procedure. One creator cannot have two procedures with the same name.

proc_defn The command that was used to create the procedure.

remarks A comment string.

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 This column contains the original source for the procedure if the preserve_source_format option is ON. It is used to maintain the appearance of the original text. For more information, see "PRESERVE_SOURCE_FORMAT option [database]" [*ASA Database Administration Guide,* page 683].

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.

SYSPROCPARM system table

Column name	Column type	Column constraint	Table constraint
proc_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSPROCEDURE.
parm_id	SMALLINT	NOT NULL	Primary key.
parm_type	SMALLINT	NOT NULL	
parm_mode_in	CHAR(1)	NOT NULL	
parm_mode_out	CHAR(1)	NOT NULL	
domain_id	SMALLINT	NOT NULL	Foreign key references SYSDOMAIN.
width	SMALLINT	NOT NULL	
scale	SMALLINT	NOT NULL	
parm_name	CHAR(128)	NOT NULL	
remarks	LONG VARCHAR		
”default”	LONG VARCHAR		
user_type	INTEGER		

Each parameter to a procedure in the database is described by one row in SYSPROCPARM.

proc_id Uniquely identifies the procedure to which this 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 return result sets

- **2** SQLSTATE error value
- **3** SQLCODE error value
- **4** Return value from function

parm_mode_in (Y/N) Indicates whether this parameter supplies a value to the procedure (**IN** or **INOUT** parameters).

parm_mode_out (Y/N) Indicates whether this 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 table.

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 The number of digits after the decimal point for numeric data type parameters, and zero for all other data types.

parm_name The name of the procedure parameter.

remarks A comment string.

default Unused.

user_type The user type of the parameter.

SYSPROCPERM system table

Column name	Column type	Column constraint	Table constraints
proc_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSPROCEDURE.
grantee	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.user_id.

Only users who have been granted permission can call a procedure. Each row of the SYSPROCPERM table corresponds to one user granted permission to call one procedure.

proc_id The procedure number uniquely identifies the procedure for which permission has been granted.

grantee The user number of the user ID receiving the permission.

SYSPUBLICATION system table

Column name	Column type	Column constraint	Table constraints
publication_id	UNSIGNED INT	NOT NULL	Primary key.
creator	UNSIGNED INT	NOT NULL	Unique index. Foreign key references SYSUSERPERM.user_id.
publication_name	CHAR(128)	NOT NULL	Unique index.
remarks	LONG VARCHAR		
type	CHAR(1)	NOT NULL	

Each row describes a SQL Remote or MobiLink publication.

publication_id A unique identifying number for the publication.

creator The owner of the publication.

publication_name The name of the publication.

remarks Descriptive comments.

type This column is deprecated.

SYSREMOTEOPTION system table

Function

Each row describes the values of a SQL Remote message link parameter.

Columns

Column	Data type	Column Constraint	Table constraints
option_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references to SYSREMOTEOPTIONTYPE.
user_id	UNSIGNED INT	NOT NULL	Primary key.
”setting”	VARCHAR(255)	NOT NULL	

Some columns in this table contain potentially sensitive data. For that reason, access to this table is restricted to users with DBA authority. The SYSREMOTEOPTION2 view provides public access to the data in this table 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.

SYSREMOTEOPTIONTYPE system table

Function

Each row describes one of the SQL Remote message link parameters.

Columns

Column	Data type	Column constraint	Table constraints
option_id	UNSIGNED INT	NOT NULL	Primary key.
type_id	UNSIGNED INT	NOT NULL	Foreign key references SYSREMOTETYPE.
”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 this parameter.

option The name of the message link parameter.

SYSREMOTETYPE system table

Column name	Column type	Column constraint	Table constraints
type_id	SMALLINT	NOT NULL	Primary key.
type_name	CHAR(128)	NOT NULL	Unique index.
publisher_address	LONG VARCHAR	NOT NULL	
remarks	LONG VARCHAR		

The SYSREMOTETYPE system table contains information about SQL Remote.

type_id Identifies which of the message systems supported by SQL Remote is to be used to send messages to this user.

type_name The name of the message system supported by SQL Remote.

publisher_address The address of the remote database publisher.

remarks Descriptive comments.

SYSREMOTEUSER system table

Column name	Column type	Column constraint	Table constraints
user_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.
consolidate	CHAR(1)	NOT NULL	
type_id	SMALLINT	NOT NULL	Foreign key references SYSREMOTE-TYPE.
address	LONG VARCHAR	NOT NULL	
frequency	CHAR(1)	NOT NULL	Unique index.
send_time	TIME		Unique index.
log_send	NUMERIC(20,0)	NOT NULL	
time_sent	TIMESTAMP		
log_sent	NUMERIC(20,0)	NOT NULL	
confirm_sent	NUMERIC(20,0)	NOT NULL	
send_count	INTEGER	NOT NULL	
resend_count	INTEGER	NOT NULL	
time_received	TIMESTAMP		
log_received	NUMERIC(20,0)	NOT NULL	
confirm_received	NUMERIC(20,0)		
receive_count	INTEGER	NOT NULL	
rereceive_count	INTEGER	NOT NULL	

Each row 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.

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 this 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.

SYSSCHEDULE system table

Column name	Column type	Column constraint	Table constraints
event_id	INTEGER	NOT NULL	Primary key. Foreign key references SYSEVENT.
sched_name	VARCHAR(128)	NOT NULL	Primary key.
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		

Each row in SYSSCHEDULE describes the times at which an event is to fire, as specified by the SCHEDULE clause of CREATE EVENT.

event_ id The unique number assigned to each event.

sched_name The name associated with a schedule.

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:

- 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.

SYSSERVERS system table

Column name	Column type	Column constraint	Table Constraints
srvid	INTEGER	NOT NULL	Primary key.
srvname	VARCHAR(128)	NOT NULL	
srvclass	LONG VARCHAR	NOT NULL	
srvinfo	LONG VARCHAR		
srvreadonly	CHAR(1)	NOT NULL	

Each row describes a remote server.

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.

SYSSQLSERVERTYPE system table

Column name	Column type	Column constraint	Table constraints
ss_user_type	SMALLINT	NOT NULL	Primary key.
ss_domain_id	SMALLINT	NOT NULL	
ss_type_name	VARCHAR (30)	NOT NULL	
primary_sa_domain_id	SMALLINT	NOT NULL	
primary_sa_user_type	SMALLINT		

This table contains information relating to compatibility with Adaptive Server Enterprise.

ss_user_type The Adaptive Server Enterprise user type.

ss_domain_id The Adaptive Server Enterprise domain id.

ss_type_name Contains the Adaptive Server Enterprise type name.

primary_sa_domain_id The Adaptive Server Anywhere primary domain id.

primary_sa_user_type A UNSIGNED INT field containing the Adaptive Server Anywhere primary user type.

SYSSUBSCRIPTION system table

Column name	**Column type**	**Column constraint**	**Table constraints**
publication_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSPUBLICATION.
user_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.
subscribe_by	CHAR(128)	NOT NULL	Primary key.
created	NUMERIC(20,0)	NOT NULL	
started	NUMERIC(20,0)		

Each row describes a subscription from one user ID (which must have REMOTE permissions) to one publication.

publication_id The identifier for the publication to which the user ID is subscribed.

user_id The user number that 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.

SYSSYNC system table

Column name	Column type	Column constraint	Table constraints
sync_id	UNSIGNED INT	NOT NULL	Primary key.
type	CHAR(1)	NOT NULL	
publication_id	UNSIGNED INT		Foreign key references SYSPUBLICATION.
progress	NUMERIC(20,0)		
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 default ‘jan-1-1900’	
created	NUMERIC(20, 0)		
log_sent	NUMERIC(20,0)		
generation_number	INTEGER	NOT NULL default 0	
extended_state	VARCHAR(1024)	NOT NULL default ‘’	

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.

sync_id A SMALLINT field uniquely identifying the row.

type A CHAR(1) field describing the type of synchronization object: ‘D’ means definition, ‘T’ means template, and ‘S’ means site.

publication_id A publication_id found in the SYSPUBLICATION table.

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 synchronization 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 synchronization server.

last_upload_time Indicates the last time (measured at the MobiLink synchronization server) that information was successfully uploaded.

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.

extended_state Reserved for internal use.

SYSTABLE system table

Column name	Column type	Column constraint	Table constraints
table_id	UNSIGNED INT	NOT NULL	Primary key.
file_id	SMALLINT	NOT NULL	Foreign key references SYSFILE.
count	UNSIGNED BIGINT	NOT NULL	
first_page	INTEGER	NOT NULL	
last_page	INTEGER	NOT NULL	
primary_root	INTEGER	NOT NULL	
creator	UNSIGNED INT	NOT NULL	Unique index. Foreign key references SYSUSERPERM.-user_id.
first_ext_page	INTEGER	NOT NULL	
last_ext_page	INTEGER	NOT NULL	
table_page_count	INTEGER	NOT NULL	
ext_page_count	INTEGER	NOT NULL	
table_name	CHAR(128)	NOT NULL	Unique index.
table_type	CHAR(10)	NOT NULL	
view_def	LONG VARCHAR		
remarks	LONG VARCHAR		
replicate	CHAR(1)	NOT NULL	
existing_obj	CHAR(1)		
remote_location	LONG VARCHAR		
remote_objtype	CHAR(1)		

Column name	**Column type**	**Column constraint**	**Table constraints**
srvid	INTEGER		Foreign key references SYSSERVERS.
server_type	CHAR(4)	NOT NULL	
primary_hash_limit	SMALL INT	NOT NULL	
page_map_start	INTEGER	NOT NULL	
source	LONG VARCHAR		

Each row of SYSTABLE describes one table or view in the database.

table_id Each table or view is assigned a unique number (the table number) which is the primary key for SYSTABLE.

file_id Indicates which database file contains the table.

count The number of rows in the table is updated during each successful checkpoint. This number is used by Adaptive Server Anywhere when optimizing database access. The **count** is always 0 for a view.

first_page Each database is divided into a number of fixed-size pages. This value identifies the first page that contains information for this table, and is used internally to find the start of this table. The first_page is always 0 for a view.

last_page The last page that contains information for this table. The last_page is always 0 for a view. For global temporary tables, 0 indicates that the table was created using ON COMMIT PRESERVE ROWS while 1 indicates that the table was created using ON COMMIT DELETE ROWS.

primary_root The **primary_root** locates the root page of the index for the primary key for the table. It will be 0 for a view and for a table with no primary key.

creator The user number of the owner of the table or view.

first_ext_page The first page used for storing row extensions and blobs.

last_ext_page The last page used for storing row extensions and blobs. The pages are maintained as a doubly-linked list.

table_page_count The total number of main pages used by this table.

ext_page_count The total number of extension (blob) pages used by this table.

table_name The name of the table or view. One creator cannot have two tables or views with the same name.

table_type This column is **BASE** for base tables, **VIEW** for views, and **GBL TEMP** for global temporary tables. No entry is created for local temporary tables.

view_def For a view, this column contains the CREATE VIEW command that was used to create the view.

remarks A comment string.

replicate (Y/N) Indicates whether the table is a primary data source in a Replication Server installation.

existing_obj (Y/N) Indicates whether the table previously existed or not.

remote_location Indicates the storage location of the remote object.

remote_objtype Indicates the type of remote object: **T** if table; **V** if view; **R** if rpc; **B** if JavaBean.

srvid The unique ID for the server if this is a proxy table.

server_type The location of the data for the table. It is either SA or OMNI.

primary_hash_limit The hash size for the primary key index for this table.

page_map_start The start of the page map maintained for this table. Page maps are used to facilitate blocked I/O during sequential scans.

source This column contains the original source for a view if the preserve_source_format option is ON. It is used to maintain the appearance of the original text. For more information, see "PRESERVE_SOURCE_FORMAT option [database]" [*ASA Database Administration Guide,* page 683].

SYSTABLEPERM system table

Column name	Column type	Column constraint	Table constraints
stable_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSTABLE.table_id.
grantee	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.user_id.
grantor	UNSIGNED INT	NOT NULL	Primary key. Foreign key references SYSUSERPERM.user_id.
ttable_id	UNSIGNED INT	NOT NULL	Foreign key references SYSTABLE.table_id.
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	

Permissions given by the GRANT command are stored in SYSTABLEPERM. Each row in this table corresponds to one table, one user ID granting the permission (**grantor**) and one user ID granted the permission (**grantee**).

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 (with grant options).

Permissions
The grantee might have been given permission for the same table by another grantor. If so, this information would be recorded in a different row of SYSTABLEPERM.

stable_id The table number of the table or view to which the permissions apply.

grantor The user number of the user ID granting the permission.

grantee The user number of the user ID receiving the permission.

ttable_id In the current version of Adaptive Server Anywhere, this table number is always the same as stable_id.

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 table. If updatecols has the value Y, there will be one or more rows in SYSCOLPERM granting update permission for the columns in this table.

alterauth (Y/N/G) Indicates whether ALTER permission has been granted.

referenceauth (Y/N/G) Indicates whether REFERENCE permission has been granted.

SYSTRIGGER system table

Column name	Column type	Column constraint	Table constraints
trigger_id	UNSIGNED INT	NOT NULL	Primary key.
table_id	UNSIGNED INT	NOT NULL	Foreign key references SYSTABLE.-table_id.
event	CHAR(1)	NOT NULL	Unique.
trigger_time	CHAR(1)	NOT NULL	Unique.
trigger_order	SMALLINT		Unique.
foreign_table_id	UNSIGNED INT		Unique. Foreign key references SYSFOREIGNKEY.
foreign_key_id	SMALLINT		Unique. Foreign key references SYSFOREIGNKEY.
referential_action	CHAR(1)		
trigger_name	CHAR(128)		Unique.
trigger_defn	LONG VARCHAR	NOT NULL	
remarks	LONG VARCHAR		
source	LONG VARCHAR		

Each trigger in the database is described by one row in SYSTRIGGER. The table also contains triggers that are automatically created for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE).

trigger_id Each trigger is assigned a unique number (the **trigger number**), which is the primary key for SYSTRIGGER.

table_id The table number uniquely identifies 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
- **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
- **B** BEFORE

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 A comment string.

source This column contains the original source for the trigger if the preserve_source_format option is ON. It is used to maintain the appearance of the original text. For more information, see "PRESERVE_SOURCE_FORMAT option [database]" [*ASA Database Administration Guide*, page 683].

SYSTYPEMAP system table

Column name	Column type	Column constraint	Table constraints
ss_user_type	SMALLINT	NOT NULL	
sa_domain_id	SMALLINT	NOT NULL	Foreign key references SYSDOMAIN.
sa_user_type	SMALLINT		
nullable	CHAR(1)		

The SYSTYPEMAP system table contains the compatibility mapping values for the SYSSQLSERVERTYPE system table.

ss_user_type Contains the Adaptive Server Enterprise user type.

sa_domain_id Contains the Adaptive Server Anywhere domain_id.

sa_user_type Contains the Adaptive Server Anywhere user type.

nullable This field describes whether or not the type can be null.

SYSUSERMESSAGES system table

Column name	**Column type**	**Column constraint**	**Table constraints**
error	INTEGER	NOT NULL	Unique.
uid	UNSIGNED INT	NOT NULL	
description	VARCHAR(255)	NOT NULL	
langid	SMALLINT	NOT NULL	Unique.

Each row holds a user-defined message for an error condition.

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.

SYSUSERPERM system table

Column name	Column type	Column constraint	Table constraints
user_id	UNSIGNED INT	NOT NULL	Primary key.
user_name	CHAR(128)	NOT NULL	Unique index.
password	BINARY(36)		
resourceauth	CHAR(1)	NOT NULL	
dbaauth	CHAR(1)	NOT NULL	
scheduleauth	CHAR(1)	NOT NULL	
publishauth	CHAR(1)	NOT NULL	
remotedbaauth	CHAR(1)	NOT NULL	
user_group	CHAR(1)	NOT NULL	
remarks	LONG VARCHAR		

DBA permissions required
SYSUSERPERM contains passwords, so DBA permissions are required to SELECT from it.

Each row of SYSUSERPERM describes one user ID.

user_id Each new user ID is assigned a unique number, which is the primary key for SYSUSERPERM.

user_name A string containing a unique name for the user ID.

password The password for the user ID. The password contains the NULL value for the special user IDs **SYS**, **PUBLIC**, and **dbo**. This prevents anyone from connecting to these user IDs.

resourceauth (Y/N) Indicates whether the user has RESOURCE authority. Resource authority is required to create tables.

dbaauth (Y/N) Indicates whether the user has DBA (database administrator) authority. DBA authority is very powerful, and should be restricted to as few user IDs as possible for security purposes.

scheduleauth (Y/N) Indicates whether the user has SCHEDULE authority. This is currently not used.

publishauth (Y/N) Indicates whether the user has the SQL Remote publisher authority.

remotedbaauth (Y/N) Indicates whether the user has the SQL Remote remote DBA authority.

user_group (Y/N) Indicates whether the user is a group.

remarks A comment string.

When a database is initialized, the following user IDs are created:

- **SYS** The creator of all the system tables.
- **PUBLIC** A special user ID used to record PUBLIC permissions.
- **DBA** The database administrator user ID is the only usable user ID in an initialized system. The initial password is SQL.
- **dbo** The owner of many of the system stored procedures and Transact-SQL capability views.

There is no way to connect to the **SYS**, **PUBLIC**, or **dbo** user IDs.

SYSUSERTYPE system table

Column name	Column type	Column constraint	Table constraints
type_id	SMALLINT	NOT NULL	Primary key.
creator	UNSIGNED INT	NOT NULL	Foreign key references SYSUSERPERM.user_id.
domain_id	SMALLINT	NOT NULL	Foreign key references SYSDOMAIN.
nulls	CHAR(1)	NOT NULL	
width	SMALLINT	NOT NULL	
scale	SMALLINT	NOT NULL	
type_name	CHAR(128)	NOT NULL	Unique.
”default”	LONG VARCHAR		
”check”	LONG VARCHAR		
format_str	CHAR(128)		
super_type_id	SMALLINT		Foreign key references SYSUSERTYPE.type_id.

Each row holds a description of a user-defined data type.

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 table.

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.

format_str Currently unused.

SYSWEBSERVICE system table

Column name	Column type	Column constraint	Table constraints
service_id	UNSIGNED INT	NOT NULL	Primary key.
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		

Each row holds a description of a web service.

service_id A unique identifying number for the web service.

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.

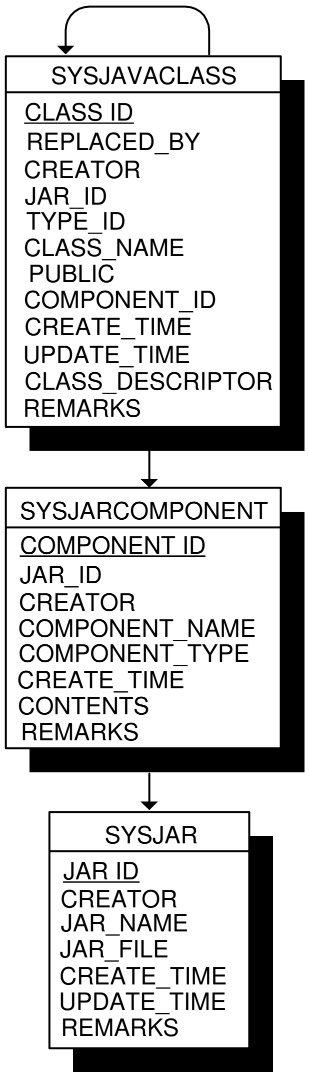
statement A SQL statement that is always executed in response to a request. If NULL, arbitrary statements contained in each request are executed instead. Ignored for services of type DISH.

Other system tables

Following is information about system tables used by Java in the database and SQL Remote.

Java system tables

The system tables that are used for Java in the database 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 System Tables" [*MobiLink Administration Guide*, page 501].

SQL Remote system tables

☞ For information about the SQL Remote system tables, see "SQL

Remote system tables” [*SQL Remote User’s Guide*, page 326].

CHAPTER 6

System Views

About this chapter

This chapter lists predefined views for the system tables.

Contents

Introduction

The system tables described in System Tables use numbers to identify tables, user IDs, and so forth. While this is efficient for internal use, it makes these tables difficult for people to interpret. A number of predefined system views are provided that present the information in the system tables in a more readable format.

System view definitions

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.

SYSARTICLECOLS system view

View name	Base table	Remarks
SYSARTICLECOLS	“SYSARTICLECOL system table” on page 677	

SYSARTICLES system view

View name	Base table	Remarks
SYSARTICLES	“SYSARTICLE system table” on page 676	

SYSCAPABILITIES system view

View name	Base tables	Remarks
SYSCAPABILITIES	"SYSCAPABILITY system table" on page 682 "SYSCAPABILITYNAME system table" on page 683	

SYSCATALOG system view

View name	Base table	Remarks
SYSCATALOG	"SYSTABLE system table" on page 742	

SYSCOLAUTH system view

View name	Base table	Remarks
SYSCOLAUTH	"SYSCOLPERM system table" on page 687	Presents column update information from SYSCOLPERM in a more readable format.

SYSCOLSTATS system view

View name	Base table	Remarks
SYSCOLSTATS	"SYSCOLSTAT system table" on page 688	

SYSCOLUMNS system view

View name	Base table	Remarks
SYSCOLUMNS	"SYSCOLUMN system table" on page 689	

SYSFOREIGNKEYS system view

View name	Base tables	Remarks
SYSFOREIGNKEYS	"SYSFOREIGNKEY system table" on page 701 "SYSFKCOL system table" on page 700	

SYSGROUPS system view

View name	Base table	Remarks
SYSGROUPS	"SYSGROUP system table" on page 703	

SYSINDEXES system view

View name	Base tables	Remarks
SYSINDEXES	"SYSINDEX system table" on page 706 "SYSIXCOL system table" on page 710	

SYSOPTIONS system view

View name	Base table	Remarks
SYSOPTIONS	"SYSOPTION system table" on page 717	

SYSOPTORDERS system view

This view is reserved for system use.

SYSOPTPLANS system view

This view is reserved for system use.

SYSOPTSTRATEGIES system view

This view is reserved for system use.

SYSPROCAUTH system view

View name	Base table	Remarks
SYSPROCAUTH	"SYSPROCPERM system table" on page 728	Presents procedure permissions information from SYSPROCPERM in a more readable format.

SYSPROCPARMS system view

View name	Base table	Remarks
SYSPROCPARMS	"SYSPROCPARM system table" on page 726	Presents the procedure parameters from SYSPROCPARM in a more readable format.

SYSPUBLICATIONS system view

View name	Base tables	Remarks
SYSPUBLICATIONS	"SYSUSERPERM system table" on page 752 "SYSPUBLICATION system table" on page 729	Presents the user name from the SYSUSERPERM table for all creators and displays the publication name and remarks from the SYSPUBLICATION table in a more readable format.

SYSREMOTEOPTION2 system view

View name	Base tables	Remarks
SYSREMOTEOPTION2	"SYSREMOTEOPTION system table" on page 730 See the "SYSREMOTEOPTIONTYPE system table" on page 731	Presents the columns from SYSREMOTEOPTION and SYSREMOTEOPTIONTYPE that do not contain sensitive data in a more readable format.

SYSREMOTEOPTIONS system view

View name	Base tables	Remarks
SYSREMOTEOPTIONS	"SYSREMOTEOPTION system table" on page 730 For more information see "SYSREMOTEOPTIONTYPE system table" on page 731	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.

SYSREMOTETYPES system view

View name	Base table	Remarks
SYSREMOTETYPES	"SYSREMOTETYPE system table" on page 732	Presents the remote types from the SYSREMOTETYPE table.

SYSREMOTEUSERS system view

View name	Base table	Remarks
SYSREMOTEUSERS	"SYSREMOTEUSER system table" on page 733	

SYSSUBSCRIPTIONS system view

View name	Base table	Remarks
SYSSUBSCRIPTIONS	"SYSPUBLICATION system table" on page 729	Presents subscription information, such as the publication name, creation time, and start time from the SYSPUBLICATION table in a more readable format.

SYSSYNCDEFINITIONS system view (deprecated)

A view of synchronization definitions for MobiLink synchronization. This view is deprecated.

SYSSYNCPUBLICATIONDEFAULTS system view

View name	Base table	Remarks
SYSSYNCPUBLICATIONDEFAULTS	"SYSSYNC system table" on page 740 "SYSPUBLICATION system table" on page 729	A view of default synchronization settings associated with publications involved in MobiLink synchronization.

SYSSYNC2 system view

View name	Base table	Remarks
SYSSYNC2	"SYSSYNC system table" on page 740	A union of the columns in SYSSYNCPUBLICATIONDEFAULTS, SYSSYNCUSERS, and SYSSYNCSUBSCRIPTIONS that do not contain sensitive data.

SYSSYNCS system view

View name	Base tables	Remarks
SYSSYNCS	"SYSSYNC system table" on page 740 "SYSPUBLICATION system table" on page 729	A union of SYSSYNCPUBLICATIONDEFAULTS, SYSSYNCUSERS and SYSSYNCSUBSCRIPTIONS. 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 insensitive data.

SYSSYNCSUBSCRIPTIONS system view

View name	Base table	Remarks
SYSSYNCSUBSCRIPTIONS	"SYSSYNC system table" on page 740 "SYSPUBLICATION system table" on page 729	A view of synchronization settings associated with MobiLink synchronization subscriptions.

SYSSYNCUSERS system view

View name	Base table	Remarks
SYSSYNCUSERS	"SYSSYNC system table" on page 740	A view of synchronization settings associated with MobiLink synchronization users.

SYSTABAUTH system view

View name	Base table	Remarks
SYSTABAUTH	"SYSTABLEPERM system table" on page 745	

SYSTRIGGERS system view

View name	Base table	Remarks
SYSTRIGGERS	"SYSTRIGGER system table" on page 747	

SYSUSERAUTH system view

View name	Base table	Remarks
SYSUSERAUTH	"SYSUSERPERM system table" on page 752	Presents all the information in the table SYSUSERPERM except for user numbers. Because this view displays passwords, this system view does not have PUBLIC select permission. (All other system views have PUBLIC select permission.)

SYSUSERLIST system view

View name	Base table	Remarks
SYSUSERAUTH	"SYSUSERPERM system table" on page 752	Presents all the information in SYSUSERAUTH except passwords.

SYSUSEROPTIONS system view

View name	Base table	Remarks
SYSUSEROPTIONS	“SYSOPTIONS system view” on page 764	Presents permanent 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.

SYSUSERPERMS system view

View name	Base table	Remarks
SYSUSERPERMS	“SYSUSERPERM system table” on page 752	Contains exactly the same information as the table SYSUSERPERM, except the password is omitted. All users have read access to this view, but only the DBA has access to the underlying table (SYSUSERPERM).

SYSVIEWS system view

View name	Base table	Remarks
SYSVIEWS	“SYSTABLE system table” on page 742 “SYSUSERPERM system table” on page 752	Lists views along with their definitions.

Views for Transact-SQL compatibility

Adaptive Server Enterprise and Adaptive Server Anywhere have different system catalogs, reflecting the different uses for the two products.

In Adaptive Server Enterprise, a single master database contains a set of system tables with information that applies to all databases on the server. Many databases may exist within the master database, and each has additional system tables associated with it.

In Adaptive Server Anywhere, each database exists independently, and contains its own system tables. There is no master database that contains system information on a collection of databases. Each server may run several databases at a time, dynamically loading and unloading each database as needed.

The Adaptive Server Enterprise and Adaptive Server 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 Adaptive Server Anywhere system tables and views are owned by the special user SYS and exist separately in each database.

To assist in preparing compatible applications, Adaptive Server Anywhere provides a set of views owned by the special user *dbo*, which correspond to the Adaptive Server Enterprise system tables and views. Where architectural differences make the contents of a particular Adaptive Server Enterprise table or view meaningless in a Adaptive Server Anywhere context, the view is empty, containing just the column names and data types.

The following tables list the Adaptive Server Enterprise system tables and their implementation in the Adaptive Server Anywhere system catalog. The owner of all tables is *dbo* in each DBMS.

Tables existing in each Adaptive Server Enterprise database

Table name	Description	Data?
sysalternates	One row for each user mapped to a database user	No
syscolumns	One row for each column in a table or view, and for each parameter in a procedure	Yes
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving the SQL definition statement	Yes

Table name	Description	Data?
sysconstraints	One row for each referential or check constraint associated with a table or column	No
sysdepends	One row for each procedure, view, or table that is referenced by a procedure, view, or trigger	No
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.	Yes
syskeys	One row for each primary, foreign, or common key; set by the user (not maintained by Adaptive Server Enterprise)	No
syslogs	Transaction log	No
sysobjects	One row for each table, view, procedure, rule, trigger default, log, or (in tempdb only) temporary object	Contains compatible data only
sysprocedures	One row for each view, rule, default, trigger, or procedure, giving the internal definition	No
sysprotects	User permissions information	No
sysreferences	One row for each referential integrity constraint declared on a table or column	No
sysroles	Maps server-wide roles to local database groups	No
syssegments	One row for each segment (named collection of disk pieces)	No
systhresholds	One row for each threshold defined for the database	No
systypes	One row for each system-supplied or user-defined data type	Yes
sysusermessages	One row for each user-defined message	Yes (this is an Adaptive Server Anywhere system table)

Table name	Description	Data?
sysusers	One row for each user allowed in the database	Yes

Tables existing in the Adaptive Server Enterprise master database

Table name	Description	Data?
syscharsets	One row for each character set or sort order	No
sysconfigures	One row for each user-settable configuration parameter	No
syscurconfigs	Information about configuration parameters currently being used by the server	No
sysdatabases	One row for each database on the server	No
sysdevices	One row for each tape dump device, disk dump device, disk for databases, or disk partition for databases	No
sysengines	One row for each server currently online	No
syslanguages	One row for each language (except U.S. English) known to the server	No
syslocks	Information about active locks	No
sysloginroles	One row for each server login that possesses a system-defined role	No
syslogins	One row for each valid user account	Yes
sysmessages	One row for each system error or warning	No
sysprocesses	Information about server processes	No
sysremotelogins	One row for each remote user	No
sysservers	One row for each remote server	No
syssrvroles	One row for each server-wide role	No
sysusages	One row for each disk piece allocated to a database	No

Tables existing in the Adaptive Server Enterprise sybsecurity database

Table name	Description	Data?
sysaudits	One row for each audit record	No
sysauditoptions	One row for each global audit option	No

CHAPTER 7

System Procedures and Functions

About this chapter

This chapter documents the system-supplied stored procedures in Adaptive Server Anywhere databases. The chapter also documents system-supplied extended procedures, including procedures for sending e-mail messages.

Contents

System procedure overview

Adaptive Server Anywhere includes the following kinds of system procedures:

- Catalog stored procedures, for displaying system information in tabular form.
- Extended stored procedures for e-mail support and other functions.
- Miscellaneous procedures for controlling server behavior
- Transact-SQL system and catalog procedures.

  ☞ For a list of these system procedures see “Adaptive Server Enterprise system and catalog procedures” on page 861.
- System functions that are implemented as stored procedures.

  ☞ For information see “System functions” on page 102.

This chapter documents the first three types of procedures listed above.

System procedure and function 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 and catalog stored procedures

System and catalog stored 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.

sa_audit_string system procedure

Function	Adds a string to the transaction log.
Syntax	**sa_audit_string (** *log-string* **)**
Arguments	♦ ***log-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]" [*ASA Database Administration Guide*, page 637] ♦ "Auditing database activity" [*SQL Anywhere Studio Security Guide*, page 9]
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

Function	Checks for outstanding referential integrity violations before a commit.
Syntax	**sa_check_commit (** *out_table_name*, *out_key_name* **)**
Arguments	♦ ***out_table_name*** A varchar(128) parameter containing the name of a table with a row that is currently violating referential integrity. ♦ ***out_key_name*** 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

DBA authority required

Side effects

None

See also

- "WAIT_FOR_COMMIT option [database]" [*ASA Database Administration Guide*, page 702]
- "CREATE TABLE statement" on page 407

Example

The following set of commands can be executed from Interactive SQL. They delete rows from the department table in the sample database, in such a way as to violate referential integrity. The call to sa_check_commit checks which tables and keys have outstanding violations, and the rollback cancels the change:

```
SET TEMPORARY OPTION WAIT_FOR_COMMIT='ON'
go
DELETE FROM department
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_conn_activity system procedure

Function

Returns the most recently-prepared SQL statement for each connection to databases on the server.

Syntax

sa_conn_activity ( )

Result sets

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.
LastStatement	long varchar	The most recently-prepared SQL statement for the connection.

Remarks

The sa_conn_activity 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 turned on 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 what SQL statement is prepared for each connection. This feature can be used as an alternative to request logging.

☞ For information on the LastStatement property these values are derived from, see "Connection-level properties" [*ASA Database Administration Guide*, page 713].

Permissions

None

Side effects

None

See also

- "-zl server option" [*ASA Database Administration Guide*, page 166]
- "sa_server_option system procedure" on page 830

sa_conn_compression_info system procedure

Function

Summarizes compression rates.

Syntax

sa_conn_compression_info ([*connection-id*] **)**

Arguments

♦ ***connection-id*** This optional integer parameter specifies the ID number of a connection.

Result sets

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.

Column name	Data type	Description
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.
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.

Column name	Data type	Description
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, sa_conn_compression_info 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" [*ASA Database Administration Guide*, page 713].

Permissions

None

Side effects

None

Example

The following example uses the sa_conn_compression_info stored 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

Function

Reports connection property information.

Syntax

sa_conn_info ([*connection-id*] **)**

Arguments

- ***connection-id*** This optional integer parameter specifies the ID number of a connection.

Result sets

Column name	Data type	Description
Number	integer	The ID number of the connection.
Name	varchar(255)	The name of the current 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 Adaptive Server Anywhere, or **local** for a same-machine connection.
NodeAddr	varchar(255)	The node for the client in a client/server connection.
LastIdle	integer	The number of ticks between requests.
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.
LockName	unsigned bigint	A 64-bit unsigned integer value representing the lock for which a connection is waiting.

Column name	Data type	Description
UncmtOps	integer	The number of uncommitted operations.

Remarks

If you specify the connection ID number, sa_conn_compression_info procedure returns a result set consisting of connection 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.

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 procedure can be used to display the locks held by the blocking connection; and if a connection holds locks on several tables you can match the LockName value between sa_locks and sa_conn_info.

☞ For information on these properties, see "Connection-level properties" [*ASA Database Administration Guide*, page 713].

Permissions

None

Side effects

None

Example

The following example uses the sa_conn_info stored 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_properties system procedure

Function

Reports connection property information.

Syntax

sa_conn_properties ([*connection-id*] **)**

Arguments

♦ ***connection-id*** This optional integer parameter specifies the ID number of a connection.

Result sets

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.

If no *connection-id* is supplied, properties for all connections to the current database are returned.

Permissions

None

Side effects

None

See also

- "sa_conn_properties_by_conn system procedure" on page 785
- "sa_conn_properties_by_name system procedure" on page 787
- "System functions" on page 102
- "Connection-level properties" [*ASA Database Administration Guide*, page 713]

Example

The following example uses the sa_conn_properties stored 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	...
...	...	...	...

sa_conn_properties_by_conn system procedure

Function

Reports connection property information.

Syntax

sa_conn_properties_by_conn ([*property-name*] **)**

Arguments

- ***property-name*** This optional varchar(255) parameter specifies a connection property name.

Result sets

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

This is a variant on the sa_conn_properties system procedure, and returns the same result columns. It returns results only for connection properties that match the *property-name* string. You can use wildcards in *property-name*, as the comparison uses a LIKE operator. The result set is sorted by connection number and property name.

☞ For a list of available connection properties, see "Connection-level properties" [*ASA Database Administration Guide,* page 713].

Permissions

None

Side effects

None

See also

- "sa_conn_properties system procedure" on page 784
- "Connection-level properties" [*ASA Database Administration Guide,* page 713]

Example

The following statement returns the AnsiNull option setting for all connections:

```
CALL sa_conn_properties_by_conn( 'ansinull' )
```

The following statement returns the ANSI-related option settings for all connections:

```
CALL sa_conn_properties_by_conn( 'ansi%' )
```

Note that the same result can also be obtained by using:

```
SELECT *
FROM sa_conn_properties()
WHERE propname LIKE 'ansi%'
ORDER BY number, propname
```

sa_conn_properties_by_name system procedure

Function

Reports connection property information.

Syntax

sa_conn_properties_by_name ([*connection-id*] **)**

Arguments

♦ ***connection-id*** This optional varchar(255) parameter specifies the ID number of a connection.

Result sets

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

This is a variant on the sa_conn_properties system procedure, and returns the same result columns. For sa_conn_properties_by_name, however, the information is sorted by property name and connection number.

☞ For a list of available connection properties, see "System functions" on page 102.

Note that the same result can be obtained using the following:

```
SELECT *
FROM sa_conn_properties()
ORDER BY propname, number
```

Permissions

None

Side effects

None

See also

♦ "sa_conn_properties system procedure" on page 784

- ♦ “Connection-level properties” [*ASA Database Administration Guide*, page 713]

Example

The following example uses the sa_conn_properties_by_name stored procedure to return a result set summarizing connection property information sorted by property name and connection number.

```
CALL sa_conn_properties_by_name()
```

Number	PropNum	PropName	...
46	279	Allow_nulls_-by_default	...
50	279	Allow_nulls_-by_default	...
...	...	...	...

sa_db_info system procedure

Function

Reports database property information.

Syntax

sa_db_info ([*database-id*] **)**

Arguments

- ♦ ***database-id*** This optional integer parameter specifies the ID number of a database.

Result sets

Column name	Data type	Description
Number	integer	The ID number of the connection.
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 *database-id* is supplied, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- ♦ "sa_db_properties system procedure" on page 789
- ♦ "Database-level properties" [*ASA Database Administration Guide,* page 733]

Example

The following statement returns a row for each database that is running on the server:

it returns one row for each database running

```
CALL sa_db_info()
```

Property	Value
Number	0
Alias	asademo
File	C:\Program Files\Sybase\SQL Anywhere 9\asademo.db
ConnCount	1
PageSize	2048
LogName	C:\Program Files\Sybase\SQL Anywhere 9\asademo.log

sa_db_properties system procedure

Function

Reports database property information.

Syntax

sa_db_properties ([*database-id*] **)**

Arguments

- ♦ ***database-id*** This optional integer parameter specifies the ID number of a database.

Result sets

Column name	Data type	Description
Number	integer	The ID number of the database.
PropNum	integer	The database property number.
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, sa_db_properties returns the database ID number and the PropNum, PropName, PropDescription, and Value for each available database property.

If no *database-id* is supplied, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- "sa_db_info system procedure" on page 788
- "Database-level properties" [*ASA Database Administration Guide,* page 733]

Example

The following example uses the sa_db_properties() stored 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_disable_auditing_type

Function

Disables auditing of specific events.

Syntax

sa_disable_auditing_type (*string* **)**

Arguments

- ***string*** This varchar(128) parameter is a comma-delimited string containing one or more of:

  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 Studio Security Guide*, page 9]
- "AUDITING option [database]" [*ASA Database Administration Guide*, page 637]

Example

To disable all auditing:

```
CALL sa_disable_auditing_type('all')
```

sa_disk_free_space system procedure

Function

Reports information about space available for a dbspace, transaction log, transaction log mirror, and/or temporary file.

Syntax

sa_disk_free_space ([*string*] **)**

Arguments

- ***string*** This string parameter can be one of *dbspace-name*, **log**, **mirror**, or **temp**.

  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 when a dbspace exists called log.

Result sets

Column name	Data type	Description
dbspace_name	varchar(128)	This is the dbspace name, **Transaction Log**, **Mirror Log**, or **Temporary File**.
free_space	unsigned bigint	The number of free bytes on the volume.

Remarks If the *string* 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 *string* is specified, then exactly one or zero rows will be returned (zero if no such dbspace exists, or if **log** or **mirror** is specified and there is no log or mirror file).

Permissions DBA authority required

Side effects None

Example The following example uses the sa_disk_free_space stored 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

Function Enables auditing and specifies which events to audit.

Syntax **sa_enable_auditing_type (** *string* **)**

Arguments

- ***string*** This varchar(128) parameter is a comma-delimited string containing one or more of:

  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. In order 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 Studio Security Guide*, page 9]
- "AUDITING option [database]" [*ASA Database Administration Guide*, page 637]

Example

To enable only option auditing:

```
CALL sa_disable_auditing_type('all')
```

```
CALL sa_enable_auditing_type('options')
```

sa_eng_properties system procedure

Function

Reports database server property information.

Syntax

sa_eng_properties ( )

Result sets

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.

☞ For a list of available database server properties, see "System functions" on page 102.

Permissions
None

Side effects
None

See also
- ♦ "Server-level properties" [*ASA Database Administration Guide,* page 725]

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

Function
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 of use in performance measurement to ensure repeatable results.

Permissions
DBA authority required

Side effects
None

sa_flush_statistics system procedure

Function	Saves all cost model statistics in the database server cache.
Syntax	**sa_flush_statistics ( )**
Remarks	Database administrators can use this procedure to ensure that cost model statistics in the database server cache that have been created and/or gathered, but not yet saved to disk, are flushed out with immediate effect. 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

sa_get_dtt system procedure

Function	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*** This unsigned smallint parameter specifies 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 system table SYSOPTSTAT.

Result sets

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 milliseconds, of reading one page.
WriteTime	unsigned integer	Amortized cost, in milliseconds, of writing one page.

Permissions	DBA authority required
Side effects	None

See also

- “SYSFILE system table” on page 699
- “SYSOPTSTAT system table” on page 723

sa_get_histogram system procedure

Function

Retrieves the histogram for a column.

Syntax

```
sa_get_histogram (
 column,
 table
 [, owner ]
)
```

Arguments

- ***column*** This char(128) parameter specifies the column.
- ***table*** This char(128) parameter specifies the table.
- ***owner*** This optional char(128) parameter specifies the owner.

Result sets

Column name	Data type	Description
StepNumber	smallint	Histogram bucket number. The frequency of the first bucket (StepNumber = 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 system table SYSCOLSTAT.

A singleton bucket is indicated by a Low value in the result set being equal to the corresponding High value.

☞ For more information about histograms and query optimization and execution in Adaptive Server Anywhere, see “Optimizer estimates and histograms” [*ASA SQL User’s Guide*, page 397].

It is recommended that you view histograms using the Histogram utility.

☞ For more information about the Histogram utility, see "The Histogram utility" [*ASA Database Administration Guide*, page 526].

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 SYSCOLSTAT. Attempting to retrieve string data generates an error.

Permissions

SELECT permission required

Side effects

None

See also

- "The Histogram utility" [*ASA Database Administration Guide*, page 526]
- "Optimizer estimates and histograms" [*ASA SQL User's Guide*, page 397]
- "ESTIMATE function [Miscellaneous]" on page 146
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 147
- "SYSCOLSTAT system table" on page 688

sa_get_request_profile system procedure

Function

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*** This optional long varchar parameter specifies the request logging filename.
- ***conn_id*** This optional unsigned integer parameter specifies the ID number of a connection.
- ***first_file*** This optional integer parameter specifies the first request log file to analyze.
- ***num_files*** This optional integer parameter specifies 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, 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 798
- "sa_statement_text system procedure" on page 836
- "sa_server_option system procedure" on page 830

sa_get_request_times system procedure

Function

Analyzes the request log to determine statement execution times.

Syntax

sa_get_request_times (
filename
[, *conn_id*
[, *first_file*
[, *num_files*]]
)

Arguments

- ***filename*** This optional LONG VARCHAR parameter specifies the request logging filename.
- ***conn_id*** This optional UNSIGNED INTEGER parameter specifies the ID number of a connection.
- ***first_file*** This optional INTEGER parameter specifies the first file to analyze.
- ***num_files*** This optional INTEGER parameter specifies 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 797
- "sa_statement_text system procedure" on page 836
- "sa_server_option system procedure" on page 830

sa_get_server_messages system procedure

Function

Allows you to return constants from the server messages window as a result set.

Syntax

sa_get_server_messages (*integer* **)**

Arguments

- ***integer*** This integer parameter indicates the line number to start displaying server messages.

Result sets

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 stored 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

Function

Returns HTTP header names and values.

Syntax

sa_http_header_info ([*header_parm*] **)**

Arguments

- ***header_parm*** This optional varchar(255) parameter specifies a HTTP header name.

Result sets

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 stored 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

"Using Web Services" [*ASA Database Administration Guide*, page 225]

"sa_http_variable_info system procedure" on page 801

sa_http_variable_info system procedure

Function

Returns HTTP variable names and values.

Syntax

sa_http_variable_info ([*variable_parm*] **)**

Arguments

- ***variable_parm*** This optional varchar(255) parameter specifies a HTTP variable name.

Result sets

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 stored 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

"Using Web Services" [*ASA Database Administration Guide*, page 225]

"sa_http_header_info system procedure" on page 800

sa_index_density system procedure

Function

Reports information about the amount of fragmentation within database indexes.

Syntax

sa_index_density (
[*table_name*
[, *owner_name*]]
)

Arguments

- ***table_name*** This optional char(128) parameter specifies the table name.
- ***owner_name*** This optional char(128) parameter specifies the owner name.

Result sets

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 values: ♦ **0** for primary keys ♦ **SYSFOREIGNKEY.*foreign_key_id*** for foreign keys ♦ **SYSINDEX.*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

- "Index fragmentation" [*ASA SQL User's Guide*, page 200]

Example

The following example uses the sa_index_density stored 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

Function

Assists in performance tuning by reporting the number of levels in an index.

Syntax

sa_index_levels ([*table_name* [, *owner_name*]] **)**

Arguments

- ***table_name*** This optional char(128) parameter specifies the table name.
- ***owner_name*** This optional char(128) parameter specifies the owner name.

Result sets

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: ♦ **0** for primary keys ♦ **SYSFOREIGNKEY.*foreign_-key_id*** for foreign keys ♦ **SYSINDEX.*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 con-straints ♦ **NUI** for non-unique in-dexes
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 *table_name* is supplied, levels for all indexes on that table are supplied. If *table_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 368
- ♦ "Using indexes" [*ASA SQL User's Guide*, page 167]

Example

The following example uses the sa_index_levels stored 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

Function

Lists the classes currently loaded by the database Java virtual machine.

Syntax

sa_java_loaded_classes ( )

Result sets

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" [*ASA Programming Guide*, page 89].

sa_locks system procedure

Function

Displays all locks in the database.

Syntax

sa_locks (
 [*connection*
 [, *table_name*
 [, *max_locks*]]]
)

Arguments

- ***connection*** This integer parameter specifies a connection ID. The procedure returns lock information only about the specified connection. The default value is zero, in which case information is returned about all connections.
- ***table_name*** This char(128) parameter specifies 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.

  If the *table_name* string contains a period, the value to the left of the period is interpreted as the owner name.
- ***max_locks*** This integer parameter specifies the maximum number of locks for which to return information. The default value is 1000. The value -1 means return all lock information.

Result sets

Column name	Data type	Description
connection	integer	The connection ID that has the lock.
user_id	char(128)	The user connected through connection ID.
table_name	char(128)	The table on which the lock is held.
lock_type	char(32)	The lock type is a string of characters indicating the type of lock.
lock_name	unsigned bigint	The lock name value identifying the lock. This value can be matched with sa_conn_-info output to determine the responsible locks in a blocking situation. Lock_names can be a row ID or can be NULL.

Remarks

The sa_locks procedure returns a result set containing information about all the locks in the database.

The **lock_type** field in the result set is a string of characters indicating the type of lock. For non-NULL **lock_name** values, the **lock_type** characters are:

- **S** Shared
- **E** Exclusive
- **P** Phantom
- **A** Anti-phantom

All locks listed have exactly one of S or E specified, and may also have P, A, or both. If a lock is a phantom or anti-phantom lock, a qualifier is added to the lock type. The qualifier is as follows:

- **T** The lock is with respect to a sequential scan.
- ***** The lock is with respect to all scans.
- **nnn** An index number. The lock is with respect to a particular index.

NULL lock_name

If the **lock_name** is NULL, then the row contains information about two types of lock: a schema lock, and a lock on rows. The **lock_type** values can be a combination of:

- **S** Shared schema lock
- **E** Exclusive schema lock
- **AT** Shared row lock
- **PT** Intent mode on a row lock

The schema lock means that other transactions are prevented from modifying the table schema. This schema lock can be acquired in shared (S) or exclusive (E) mode.

The row lock applies to the rows in the table. It can be acquired in shared mode or intent mode. Shared mode is represented by **lock_type** AT, and intent mode by **lock_type** PT. If acquired in shared mode, other transactions cannot modify the rows unless they acquire the lock in intent mode. However, the lock can only be acquired in shared mode if there are no uncommitted modifications to the table by other transactions.

For example, if a connection has modified a table but not yet done a commit or rollback, then sa_locks returns a NULL lock_name for the table, and a lock_type of at least SPT. S indicates a shared lock on the schema of the table and PT indicates an intent lock on the rows in the table.

☞ For more information, see "Connection-level properties" [*ASA Database Administration Guide*, page 713], and "sa_conn_info system procedure" on page 782.

Permissions DBA authority required

Side effects None

See also

- "How locking works" [*ASA SQL User's Guide*, page 135]

sa_make_object system procedure

Function Ensures that a skeletal instance of an object exists before executing an ALTER statement.

Syntax

sa_make_object (
 object-type,
 objname
 [, *owner*
 [, *tabname*] **])**

object-type:
'procedure' | **'function'** | **'view'** | **'trigger'** | **'service'**

Arguments

- ***object-type*** This char(30) parameter specifies the type of object being created.
- ***objname*** This char(128) parameter specifies the name of the object to be created.
- ***owner*** This char(128) parameter specifies 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 it specifies 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 tables 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 FUNCTION statement" on page 275
- ♦ "ALTER PROCEDURE statement" on page 278
- ♦ "ALTER TRIGGER statement" on page 302
- ♦ "ALTER VIEW statement" on page 303
- ♦ "ALTER SERVICE statement" on page 286

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_migrate system procedure

Function

Migrates a set of remote tables to an Adaptive Server Anywhere database.

Syntax

sa_migrate (*local_table_owner* ,
server_name
[, *table_name*]
[, *owner_name*]
[, *database_name*]
[, *migrate_data*]
[, *drop_proxy_tables*]
[, *migrate_fkeys*]
)

Arguments

- ♦ ***local_table_owner*** This varchar(128) parameter specifies the user on the target Adaptive Server Anywhere database who owns the migrated tables. Use the GRANT CONNECT statement to create this user. A value is required for this parameter.

  ☞ For more information, see "GRANT statement" on page 503.

- ♦ ***server_name*** This varchar(128) parameter specifies 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.

☞ For more information, see "CREATE SERVER statement" on page 392.

- ***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*** This varchar(128) parameter specifies 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*** Specifies 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.
- ***drop_proxy_tables*** Specifies 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.
- ***migrate_fkeys*** Specifies 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.

Remarks

You can use this procedure to migrate tables to Adaptive Server Anywhere from a remote Oracle, DB2, SQL Server, Adaptive Server Enterprise, Adaptive Server Anywhere, or Access 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 stored procedure calls the following stored 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 stored 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.

☞ For more information, see "CREATE SERVER statement" on page 392 and "CREATE EXTERNLOGIN statement" on page 360.

You can migrate all the tables from the remote database to an Adaptive Server Anywhere database by specifying only the *local_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 Adaptive Server Anywhere database. If tables have different owners on the remote database and you want them to have different owners on the Adaptive Server Anywhere database, then you must migrate the tables for each owner separately, specifying the *local_table_owner* and *owner_name* parameters each time you call the sa_migrate procedure. In order to use this procedure, you must have the necessary permissions to create tables for the local Adaptive Server Anywhere user.

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

- "sa_migrate_create_remote_table_list system procedure" on page 814
- "sa_migrate_create_tables system procedure" on page 816
- "sa_migrate_data system procedure" on page 817
- "sa_migrate_create_remote_fks_list system procedure" on page 813

- “sa_migrate_create_fks system procedure” on page 812
- “sa_migrate_drop_proxy_tables system procedure” on page 819
- “Migrating databases to Adaptive Server Anywhere” [*ASA SQL User’s Guide*, page 591]

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 Adaptive Server 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 Adaptive Server 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

Function

Creates foreign keys for each table listed in the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_fks (*local_table_owner* **)**

Arguments

- ***local_table_owner*** This varchar(128) parameter specifies the user on the target Adaptive Server 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 *local_table_owner* is created using the GRANT CONNECT statement. A value is required for this parameter.

  ☞ For more information, see “GRANT statement” on page 503.

Remarks

The sa_migrate_create_fks stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

This procedure creates foreign keys for each table that is listed in the dbo.migrate_remote_fks_list table. The user specified by the *local_table_owner* argument owns the foreign keys in the target database.

If the tables in the target Adaptive Server 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.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

Permissions

None

Side effects

None

See also

- "sa_migrate system procedure" on page 809
- "sa_migrate_create_remote_table_list system procedure" on page 814
- "sa_migrate_create_tables system procedure" on page 816
- "sa_migrate_data system procedure" on page 817
- "sa_migrate_create_remote_fks_list system procedure" on page 813
- "sa_migrate_drop_proxy_tables system procedure" on page 819
- "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

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 Adaptive Sever Anywhere database.

```
CALL sa_migrate_create_fks( 'local_a' )
```

sa_migrate_create_remote_fks_list system procedure

Function

Populates the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_remote_fks_list (*server_name* **)**

Arguments

- ***server_name*** This varchar(128) parameter specifies 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.

  ☞ For more information, see "CREATE SERVER statement" on page 392.

Remarks

The sa_migrate_create_remote_fks_list stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

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.

Permissions None

Side effects None

See also

- ♦ "sa_migrate system procedure" on page 809
- ♦ "sa_migrate_create_remote_table_list system procedure" on page 814
- ♦ "sa_migrate_create_tables system procedure" on page 816
- ♦ "sa_migrate_data system procedure" on page 817
- ♦ "sa_migrate_create_fks system procedure" on page 812
- ♦ "sa_migrate_drop_proxy_tables system procedure" on page 819
- ♦ "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

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

Function Populates the dbo.migrate_remote_table_list table.

Syntax **sa_migrate_create_remote_table_list (**
server_name
[, *table_name*
[, *owner_name*
[, *database_name*]]]
)

Arguments

- ***server_name*** This varchar(128) parameter specifies 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.

  ☞ For more information, see "CREATE SERVER statement" on page 392.

- ***table_name*** This varchar(128) parameter specifies 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 *table_name* and *owner_name* parameters.

- ***owner_name*** This varchar(128) parameter specifies 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 *table_name* and *owner_name* parameters

- ***database_name*** This varchar(128) parameter specifies 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

The sa_migrate_create_remote_table_list stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

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 Adaptive Server 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 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

- ♦ "sa_migrate system procedure" on page 809
- ♦ "sa_migrate_create_tables system procedure" on page 816
- ♦ "sa_migrate_data system procedure" on page 817
- ♦ "sa_migrate_create_remote_fks_list system procedure" on page 813
- ♦ "sa_migrate_create_fks system procedure" on page 812
- ♦ "sa_migrate_drop_proxy_tables system procedure" on page 819
- ♦ "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

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

Function

Creates a proxy table and base table for each table listed in the dbo.migrate_remote_table_list table.

Syntax

sa_migrate_create_tables (*local_table_owner* **)**

Arguments

- ♦ ***local_table_owner*** This varchar(128) parameter specifies the user on the target Adaptive Server Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter.

  ☞ For more information, see "GRANT statement" on page 503.

Remarks

The sa_migrate_create_tables stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

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 *local_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 Adaptive Server 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.

Permissions

None

Side effects

None

See also

- "sa_migrate system procedure" on page 809
- "sa_migrate_create_remote_table_list system procedure" on page 814
- "sa_migrate_data system procedure" on page 817
- "sa_migrate_create_remote_fks_list system procedure" on page 813
- "sa_migrate_create_fks system procedure" on page 812
- "sa_migrate_drop_proxy_tables system procedure" on page 819
- "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

Example

The following statement creates base tables and proxy tables on the target Adaptive Server Anywhere database. These tables belong to the user local_a.

```
CALL sa_migrate_create_tables( 'local_a' )
```

sa_migrate_data system procedure

Function

Migrates data from the remote database tables to the target Adaptive Server Anywhere database.

Syntax

sa_migrate_data (*local_table_owner* **)**

Arguments

- ***local_table_owner*** This varchar(128) parameter specifies the user on the target Adaptive Server Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter.

  ☞ For more information, see "GRANT statement" on page 503.

Remarks

The sa_migrate_data stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

This procedure migrates the data from the remote database to the Adaptive Server Anywhere database for tables belonging to the user specified by the *local_table_owner* argument.

When the tables on the target Adaptive Server 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.

Permissions

None

Side effects

None

See also

- "sa_migrate system procedure" on page 809
- "sa_migrate_create_remote_table_list system procedure" on page 814
- "sa_migrate_create_tables system procedure" on page 816
- "sa_migrate_create_remote_fks_list system procedure" on page 813
- "sa_migrate_create_fks system procedure" on page 812
- "sa_migrate_drop_proxy_tables system procedure" on page 819
- "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

Example

The following statement migrates data to the target Adaptive Server Anywhere database for tables that belong to the user local_a.

```
CALL sa_migrate_data( 'local_a' )
```

sa_migrate_drop_proxy_tables system procedure

Function	Drops the proxy tables that were created for migration purposes.
Syntax	**sa_migrate_drop_proxy_tables (** *local_table_owner* **)**

Arguments

- ***local_table_owner*** This varchar(128) parameter specifies the user on the target Adaptive Server Anywhere database who owns the proxy tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter.

  ☞ For more information, see "GRANT statement" on page 503.

Remarks

The sa_migrate_drop_proxy_tables stored procedure is used with the other migration stored procedures. These procedures must be executed in the following order:

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

This procedure drops the proxy tables that were created for the migration. The user who owns these proxy tables is specified by the *local_table_owner* argument.

If the migrated tables are not all owned by the same user on the target Adaptive Server Anywhere database, you must call this procedure for each user in order to drop all the 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

- "sa_migrate system procedure" on page 809
- "sa_migrate_create_remote_table_list system procedure" on page 814
- "sa_migrate_create_tables system procedure" on page 816
- "sa_migrate_data system procedure" on page 817
- "sa_migrate_create_remote_fks_list system procedure" on page 813
- "sa_migrate_create_fks system procedure" on page 812

- ♦ "Migrating databases to Adaptive Server Anywhere" [*ASA SQL User's Guide*, page 591]

Example

The following statement drops the proxy tables on the target Adaptive Server Anywhere database that belong to the user local_a.

```
CALL sa_migrate_drop_proxy_tables( 'local_a' )
```

sa_procedure_profile system procedure

Function

Reports information about the execution time for each line within procedures that have been executed in a database.

Syntax

sa_procedure_profile (
[*p_object_name*
[, *p_owner_name*
[, *p_table_name*]]]
)

Arguments

- ♦ ***p_object_name*** This optional char(128) parameter specifies the name of the stored procedure, function, event, or trigger.
- ♦ ***p_owner_name*** This optional char(128) parameter specifies the object's owner.
- ♦ ***p_table_name*** This optional char(128) parameter specifies the table associated with a trigger (the value is NULL for other object types).

Result sets

Column name	Data type	Description
object_type	char(1)	The type of object.
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).

Column name	Data type	Description
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.
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

Before you can profile your database, you must enable profiling.

☞ For more information about enabling procedure profiling, see "sa_server_option system procedure" on page 830.

The procedure accepts three optional arguments. If you specify more than one of these arguments, you must list them in the order shown (*p_object_name*, *p_owner_name*, *p_table_name*). The arguments are strings, and must be enclosed in single quotes. The server returns data for all procedures in the database if you do not include any arguments.

The result set includes information about the execution times for individual lines within procedures, and what percentage of the total procedure execution time those lines use. The DBA can use this profiling information to fine-tune slower procedures that may decrease performance. The procedure returns the same information for stored procedures, functions, events, and triggers as the Profile tab in Sybase Central.

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

By calling the procedures of interest before you begin a profiling session, you eliminate the start-up time required for procedures to load and for the database to access tables for the first time.

Permissions DBA authority required

Side effects None

See also
- "sa_server_option system procedure" on page 830
- "sa_procedure_profile_summary system procedure" on page 822

Example The following statement returns profiling information about the tr_manager trigger:

```
CALL sa_procedure_profile (p_object_name = 'tr_manager')
```

sa_procedure_profile_summary system procedure

Function Reports summary information about the execution times for all procedures that have been executed in a database.

Syntax **sa_procedure_profile_summary (**
[*p_table_name*
[, *p_owner_name*
[, *p_object_name*
[, *p_object_type*
[, *p_ordering*]]]]]
)

Arguments The procedure accepts five optional arguments:

- ***p_object_name*** This optional char(128) parameter specifies the name of the stored procedure, function, event, or trigger.
- ***p_owner_name*** This optional char(128) parameter specifies the object's owner.
- ***p_table_name*** This optional char(128) parameter specifies the table associated with a trigger (the value is NULL for other object types).
- ***p_object_type*** This optional char(1) parameter specifies the type of object to profile. It can be one of the following:
  - **P** stored procedure
  - **F** function

- **E** event
- **T** trigger
- **S** system trigger
- **C** ON UPDATE system trigger
- **D** ON DELETE system trigger

♦ ***p_ordering*** This optional char(1) parameter determines the order of columns in the result set. If no value is given, the results are listed from the longest execution time to the shortest execution time. Values and the resulting order are:

- **P** object_type, owner_name, object_name, table_name desc
- **N** object_name, owner_name, table_name, object_type desc
- **O** owner_name, object_type, object_name, table_name desc
- **T** table_name, owner_name, object_name, object_type desc
- **E** executions desc, object_name, owner_name, table_name, object_type desc

Result sets

Column name	Data type	Description
object_type	char(1)	The type of object.
object_name	char(128)	The name of the stored procedure, function, event, or trigger.
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 millisec-onds.
foreign_owner	char(128)	The database user who owns the foreign table for a system trigger.

Column name	Data type	Description
foreign_table	char(128)	The name of the foreign table for a system trigger.

Remarks

Before you can profile your database, you must enable profiling.

☞ For more information about enabling procedure profiling, see "sa_server_option system procedure" on page 830.

The procedure displays 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. The procedure returns the same information for stored procedures, functions, events, and triggers as the Profile tab in Sybase Central.

If you specify more than one argument, you must list them in the order shown (*p_object_name*, *p_owner_name*, *p_table_name*, *p_object_type*, *p_ordering*). If you specify any of these arguments, the procedure returns only rows that match the parameters; otherwise, the server returns data for all procedures in the database. Note that the argument values are strings, and must be enclosed in quotes.

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

By calling the procedures of interest before you begin a profiling session, you eliminate the start-up time required for procedures to load and for the database to access tables for the first time.

Permissions

DBA authority required

Side effects

None

See also

- “sa_server_option system procedure” on page 830
- “sa_procedure_profile system procedure” on page 820

Example

The following statement returns profiling information about all the triggers owned by the DBA on the Product table:

```
CALL sa_procedure_profile_summary
(p_owner_name = 'dba', p_table_name = 'Product', p_object_type =
         'T')
```

sa_recompile_views system procedure

Function

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 ([*integer*] **)**

Arguments

- ***integer*** A non-zero value for this optional integer parameter will suppress errors.

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.

Permissions

DBA authority required

Side effects

If any views exist for which column definitions have not been stored, an `ALTER VIEW` *`owner`*`.`*`viewname`* `RECOMPILE` statement is executed, causing an automatic commit.

See also

- “FORCE_VIEW_CREATION option [database]” [*ASA Database Administration Guide*, page 656]
- “ALTER VIEW statement” on page 303

sa_report_deadlocks system procedure

Function

Retrieves information about deadlocks from an internal buffer created by the database server.

Syntax **sa_report_deadlocks( )**

Result sets

Column name	Data type	Description
snapshotId	bigint	The deadlock instance (all rows pertaining to a particular deadlock have the same ID).
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 stored procedure.

Permissions DBA authority required

Side effects None

See also "LOG_DEADLOCKS option [database]" [*ASA Database Administration Guide*, page 665]

"Determining who is blocked" [*ASA SQL User's Guide*, page 114]

sa_reset_identity system procedure

Function

Allows the next available identity value to be set for a table. Use this to change the autoincrement value for the next row.

Syntax

sa_reset_identity (
[*table_name*
[, *owner*
[, *new_identity_value*]]]
)

Arguments

- ***table_name*** This optional varchar(128) parameter specifies the table you want to set the identity value for.
- ***owner*** This optional varchar(128) parameter specifies the owner of the table you want to set the identity value for.
- ***new_identity_value*** This optional bigint parameter specifies identifies the number from which you want to start the value counting.

Remarks

The next value generated for a row inserted into the table will be *new_identity_value* + 1.

No checking occurs on the *new_identity_value* to ensure that it does not conflict with existing rows in the table. An invalid value could cause subsequent inserts to fail.

Permissions

DBA authority required

Side effects

Causes a checkpoint to occur after the value has been updated.

Example

The following statement resets the identity value to 101:

```
CALL sa_reset_identity ('employee', 'dba', 100)
```

sa_rowgenerator system procedure

Function

Returns a result set with rows between a specified start and end value.

Syntax

sa_rowgenerator (
[*rstart*
[, *rend*
[, *rstep*]]]**)**

Arguments

- ***rstart*** This optional integer parameter specifies the starting value. The default value is 0.
- ***rend*** This optional integer parameter specifies the ending value. The default value is 100.

- ***rstep*** This optional integer parameter specifies the increment by which the sequence values are increased. The default value is 1.

Result sets

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(1255)
```

Permissions

None

Side effects

None

See also

"RowGenerator system table" on page 675

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(zip_code) AS zips_in_range
FROM sa_rowgenerator(0,99999,10000) D LEFT JOIN employee
   ON zip_code BETWEEN r1 AND r2
GROUP BY r1, r2
ORDER BY 1
```

The following example generates 10 rows of data and inserts them into the emp table:

```
INSERT INTO emp(id, salary, name)
SELECT row_num,
   CAST( rand() * 1000 AS INTEGER),
   'Mary'
FROM sa_rowgenerator(1, 10)
```

sa_send_udp system procedure

Function

Sends a UDP packet to the specified address.

Syntax

sa_send_udp (*destAddress*, *destPort*, *msg* **)**

Arguments

- ***destAddress*** This char(254) parameter can be either the host name or IP number.
- ***destPort*** This UNSIGNED SMALLINT parameter specifies the port number to use.
- ***msg*** This LONG BINARY parameter contains 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 sa_send_udp 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 User's Guide*, page 13]

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

Function Overrides a server option while the server is running.

Syntax **sa_server_option (** *option_name*, *option_value* **)**

Arguments

- ***option_name*** This char(128) parameter specifies a server option name.
- ***option_value*** This char(128) parameter specifies the new value for the server option.

Remarks Database administrators can use this procedure to override some database server options 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 if one exists.

The options that can be reset are as follows:

Option name	**Values**	**Default**	**Server option**
ConnsDisabled	ON or OFF	OFF	
LivenessTimeout	integer, in seconds	120	-tl server option
Procedure_-profiling	ON, OFF, RESET, CLEAR	OFF	
ProfileFilterConn	*connection-id*		
ProfileFilterUser	*user-id*		

Option name	**Values**	**Default**	**Server option**
QuittingTime	valid date and time		-tq server option
RememberLast-Statement	ON or OFF	OFF	-zl server option
RequestFilter-Conn	*connection-id*, -1		
RequestFilterDB	*database-id*, -1		
RequestLogFile	*Filename*		-zo server option
RequestLogging	ALL, SQL, NONE, SQL+hostvars	NONE	-zr server option
RequestLogMax-Size	*File-size*, in bytes		-zs server option
Request-LogNumFiles	integer		-zn server option

ConnsDisabled When set to ON, no other connections are allowed to any databases on the database server.

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.

☞ For more information, see "-tl server option" [*ASA Database Administration Guide*, page 158].

Procedure_profiling 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, as well as how long each line takes to execute. You can also set procedure profiling options on the Database property sheet in Sybase Central.

- **ON** enables procedure profiling for the database you are currently connected to.
- **OFF** disables procedure profiling and leaves the profiling data available for viewing.
- **RESET** returns the profiling counters to zero, without changing the ON or OFF 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 stored procedures to retrieve profiling information from the database.

☞ For more information about procedure profiling, see "Profiling database procedures" [*ASA SQL User's Guide,* page 202].

ProfileFilterConn Instruct the database server to capture profiling information for a specific connection ID.

ProfileFilterUser Instruct the database server to capture profiling information for a specific user ID.

QuittingTime Instruct the database server to shut down at the specified time.

☞ For more information, see "-tq server option" [*ASA Database Administration Guide,* page 159].

RememberLastStatement Instruct the database server to capture the most recently-prepared SQL statement for each connection to databases 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 RememberLastStatement setting using the **RememberLastStatement** property function as follows:

```
SELECT property( 'RememberLastStatement' )
```

☞ For more information, see "Server-level properties" [*ASA Database Administration Guide,* page 725] and "-zl server option" [*ASA Database Administration Guide,* page 166].

When RememberLastStatement is turned on, the following statement returns the most recently-prepared statement for the specified connection.

```
SELECT connection_property( 'LastStatement', conn_id )
```

The stored procedure sa_conn_activity 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 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 many active connections or 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 logging information. A name of NULL stops logging to file. Any backslash characters in the filename must be doubled as this is a SQL string.

☞ For more information, see "-zo server option" [*ASA Database Administration Guide*, page 168].

RequestLogging Can be **ALL**, **SQL**, **NONE**, or **SQL+hostvars**. ON and ALL are equivalent. OFF and NONE are equivalent. 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.

When you set RequestLogging to OFF, the request log file is closed.

If you select SQL, only the following types of request are recorded:

- ♦ START DATABASE
- ♦ STOP ENGINE
- ♦ STOP DATABASE
- ♦ Statement preparation
- ♦ Statement execution
- ♦ EXECUTE IMMEDIATE statements
- ♦ Option settings
- ♦ COMMIT statements
- ♦ ROLLBACK statements
- ♦ PREPARE TO COMMIT operations
- ♦ Connections
- ♦ Disconnections
- ♦ Beginnings of transactions
- ♦ DROP STATEMENT statement
- ♦ Cursor explanations
- ♦ Cursor closings
- ♦ Cursor resume
- ♦ Errors

Setting RequestLogging to SQL+hostvars outputs *both* SQL (as though you specified `('RequestLogging', 'SQL')`) *and* host variable values to the log.

You can find the current value of the RequestLogging setting using `property('RequestLogging')`.

☞ For more information, see "-zr server option" [*ASA Database Administration Guide,* page 168], and "Server-level properties" [*ASA Database Administration Guide,* page 725].

RequestLogMaxSize The maximum size of the file used to record logging information, in bytes.

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.

☞ For more information, see "-zs server option" [*ASA Database Administration Guide*, page 169].

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.

☞ For more information, see "-zn server option" [*ASA Database Administration Guide*, page 167].

Permissions

DBA authority required

Side effects

None

Example

The following statement disallows new connections to the database server.

```
CALL sa_server_option( 'ConnsDisabled', 'ON')
```

sa_set_http_header system procedure

Function

Permits a web service to set an HTTP header in the result.

Syntax

sa_set_http_header (*field-name*, *value* **)**

Arguments

- ***field-name*** This char(128) parameter specifies a string containing the name of one of the HTTP header fields.
- ***value*** This long varchar parameter specifies 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. 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_set_http_option system procedure” on page 836

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

Function Permits a web service to set an HTTP option in the result.

Syntax **sa_set_http_option (** *option-name*, *value* **)**

Arguments

- ***option-name*** This char(128) parameter specifies a string containing the name of one of the HTTP options.
- ***value*** This long varchar parameter specifies 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.

Currently only one option is supported:

- **CharsetConversion** Controls 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.

Permissions None

Side effects None

See also

- “sa_set_http_header system procedure” on page 835

sa_statement_text system procedure

Function 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 (** *select-statement* **)**

Arguments

- ***select-statement*** This long varchar parameter contains a select statement.

Remarks The *select-statement* 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 798
- "sa_get_request_profile system procedure" on page 797

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

Function

Reports information about the fragmentation of database tables.

Syntax

sa_table_fragmentation (
  [*table_name*
  [, *owner_name*]]
)

Arguments

- ***table_name*** This char(128) parameter specifies a table name.
- ***owner_name*** This char(128) parameter specifies an owner name.

Result sets

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.

Permissions	DBA authority required
Side effects	None
See also	♦ "Table fragmentation" [*ASA SQL User's Guide*, page 198] ♦ "Defragmenting tables" [*ASA SQL User's Guide*, page 199] ♦ "Rebuilding databases" [*ASA SQL User's Guide*, page 580] ♦ "REORGANIZE TABLE statement" on page 577

sa_table_page_usage system procedure

Function	Reports information about the page usage of database tables.
Syntax	**sa_table_page_usage ( )**
Result sets	

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	♦ "The Information utility" [*ASA Database Administration Guide*, page 528]

sa_validate system procedure

Function	Validates all tables in a database.

Syntax

sa_validate (
[*tbl_name*
[, *owner_name*
[, *check_type*]]] **)**

Arguments

- ♦ ***tbl_name*** This varchar(128) parameter specifies a table name. If this parameter is NULL (the default), sa_validate validates all tables.
- ♦ ***owner_name*** This varchar(128) parameter specifies an owner name. If this parameter is NULL (the default), sa_validate validates tables for all users.
- ♦ ***check_type*** When this char(10) 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:
  - **data** Validate tables using WITH DATA CHECK.
  - **index** Validate tables using WITH INDEX CHECK.
  - **full** Validate tables using WITH FULL CHECK.
  - **express** Validate tables using WITH EXPRESS CHECK.
  - **checksum** Validate database pages using checksums.

    For more information about checksum validation, see "Ensuring your database is valid" [*ASA Database Administration Guide*, page 389].

Permissions

DBA authority required

Side effects

None

Remarks

This procedure is equivalent to calling the VALIDATE TABLE statement for each table in the database.

☞ For information, see "VALIDATE TABLE statement" on page 662.

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 validates all of the tables owned by the DBA with an index check:

```
CALL sa_validate (owner_name = 'DBA', check_type = 'index')
```

sa_verify_password system procedure

Function	Validates the password of the current user.
Syntax	**sa_verify_password (** *string* **)**
Arguments	♦ ***string*** This char(128) parameter specifies 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 861

sp_login_environment system procedure

Function 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]" [*ASA Database Administration Guide*, page 666]

sp_remote_columns system procedure

Function 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** This char(128) parameter specifies a string containing the server name as specified by the CREATE SERVER statement.
- **@table_name** This char(128) parameter specifies a table name.
- **@table_owner** This optional char(128) parameter specifies the table owner.
- **@table_qualifier** This optional char(128) parameter specifies the database name.

Result sets

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.

Column name	Data type	Description
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.
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" [*ASA SQL User's Guide*, page 601]
- "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635]
- "CREATE SERVER statement" on page 392

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

Function

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** This char(128) parameter specifies identifies the server the primary table is located on. A value is required for this parameter.
- **@sp_name** This char(128) parameter specifies the table containing the primary key. A value is required for this parameter.
- **@sp_owner** This optional char(128) parameter specifies the primary table's owner.
- **@sp_qualifier** This optional char(128) parameter specifies the database containing the primary table. This parameter is optional.

Result sets

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 sp_remote_exported_keys stored procedure's 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 foreign key is always returned.

Permissions

None

Side effects

None

See also

- ♦ "CREATE SERVER statement" on page 392
- ♦ "Tables are related by foreign keys" [*Introducing SQL Anywhere Studio,* page 120]

Example

To get information about the remote tables with foreign keys on the SYSOBJECTS table, in the production database, in a server named asetest:

```
CALL sp_remote_exported_keys (
     @server_name='asetest',
     @sp_name='sysobjects',
     @sp_qualifier='production')
```

sp_remote_imported_keys system procedure

Function

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** This optional char(128) parameter specifies the server the foreign key table is located on. A value is required for this parameter.
- ♦ **@sp_name** This optional char(128) parameter specifies the table containing the foreign key. A value is required for this parameter.
- ♦ **@sp_owner** This optional char(128) parameter specifies the foreign key table's owner.
- ♦ **@sp_qualifier** This optional char(128) parameter specifies the database containing the foreign key table.

Result sets

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

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 392
- "Tables are related by foreign keys" [*Introducing SQL Anywhere Studio*, page 120]

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

Function

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** This char(128) parameter specifies the server the remote table is located on.
- **@table_name** This optional char(128) parameter specifies the remote table.
- **@table_owner** This optional char(128) parameter specifies the owner of the remote table.
- **@table_qualifier** This optional char(128) parameter specifies the name of the remote database.

Result sets

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 stored 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

Function

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** This char(128) parameter specifies the server the remote table is located on.
- **@table_name** This char(128) parameter specifies the remote table.
- **@table_owner** This char(128) parameter specifies the owner of the remote table.
- **@table_qualifier** This char(128) parameter specifies the database.
- **@with_table_type** Specifies 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 sets

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" [*ASA SQL User's Guide*, page 601]
- "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635]
- "CREATE SERVER statement" on page 392

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

Function

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 (*servername* **)**

Arguments

- ***severname*** This char(64) parameter specifies a server defined with the CREATE SERVER statement.

Remarks

This procedure displays information about a remote server's capabilities. Adaptive Server 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 are not populated until after Adaptive Server Anywhere first connects to the remote server. This information comes from SYSCAPABILITY and SYSCAPABILITYNAME. The servername specified must be the same servername used in the CREATE SERVER statement.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "Accessing Remote Data" [*ASA SQL User's Guide*, page 601]
- "Server Classes for Remote Data Access" [*ASA SQL User's Guide*, page 635]
- "CREATE SERVER statement" on page 392

Example

To display information about the remote server testasa:

```
CALL sp_servercaps ( 'testasa' )
```

sp_tsql_environment system procedure

Function

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 wish 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 840
- ♦ "LOGIN_PROCEDURE option [database]" [*ASA Database Administration Guide*, page 666].

Example

Here is the text of the sp_tsql_environment procedure:

```
CREATE PROCEDURE dbo.sp_tsql_environment()
BEGIN
  IF db_property('IQStore')='OFF' THEN
    -- ASA 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
```

System extended stored procedures

A set of system extended procedures are included in Adaptive Server 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.

The following sections describe each of the stored procedures.

Extended stored procedures for MAPI and SMTP

Adaptive Server 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 stored procedures: each procedure calls a function in an external DLL.

In order to use the MAPI or SMTP stored procedures, a MAPI or SMTP e-mail system must be accessible from the database server machine.

The stored 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

Function

Starts an e-mail session under MAPI.

Syntax

[[*variable* =] **CALL**] **xp_startmail (**
[**mail_user** = *mail-login-name*]
[, **mail_password** = *mail-password*] **)**

Arguments

- **mail_user** This long varchar parameter specifies the MAPI login name.
- **mail_password** This long varchar parameter specifies the MAPI password.

Permissions

Not supported on NetWare.

Remarks

xp_startmail is a system stored procedure that starts an e-mail 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

☞ For a list of return codes, see "MAPI return codes" on page 854.

xp_startsmtp system procedure

Function

Starts an e-mail session under SMTP.

Syntax

[[*variable* =] **CALL**] **xp_startsmtp (**
smtp_sender = *email-address***,**
smtp_server = *smtp-server*
[**, smtp_port =** *port-number*]
[**, timeout =** *timeout*] **)**

Arguments

- **smtp_sender** This long varchar parameter specifies the e-mail 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 integer parameter specifies the port number to connect to on the SMTP server. The default is 25.
- **timeout** This integer parameter specifies how long to wait, in seconds, for a response from the server before aborting the current call to xp_sendmail. The default is 60 seconds.

Permissions

Not supported on NetWare.

Remarks

xp_startsmtp is a system stored procedure that starts a mail session for a specified e-mail address by connecting to an SMTP server.

xp_startsmtp starts a connection to a server. This connection will time out. Therefore, it is recommended that you start SMTP just before you want to execute xp_sendmail.

xp_sendmail over SMTP does not support attachments.

Return codes

☞ For a list of return codes, see "SMTP return codes" on page 855.

xp_sendmail system procedure

Function

Sends an e-mail message.

Syntax

[[*variable* =] **CALL**] **xp_sendmail (**
recipient = *mail-address*
[, **subject** = *subject*]
[, **cc_recipient** = *mail-address*]
[, **bcc_recipient** = *mail-address*]
[, **"message"** = *message-body*]
[, **include_file** = *file-name*] **)**

Arguments

- ♦ **recipient** This long varchar parameter specifies the recipient mail address.
- ♦ **subject** This long varchar parameter specifies the subject field of the message.
- ♦ **cc_recipient** This long varchar parameter specifies the cc recipient field of the message.
- ♦ **bcc_recipient** This long varchar parameter specifies the cc recipient field of the message.
- ♦ **"message"** This long varchar parameter specifies the message contents.
- ♦ **include_file** This long varchar parameter specifies an attachment file.

Permissions

Must have executed xp_startmail to start an e-mail session under MAPI, or xp_startsmtp to start an e-mail session under SMTP.

Not supported on NetWare.

Remarks

xp_sendmail is a system stored procedure that sends an e-mail 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 are strings. The length of each argument is limited only by the amount of available memory on your system. The **"message"** parameter name requires double quotes around it because MESSAGE is a reserved word.

An attachment file specified by **include_file** must exist on the database server machine.

xp_sendmail over SMTP does not support attachments.

Return codes

☞ For a list of return codes, see "Return codes for MAPI and SMTP stored procedures" on page 854.

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' )
```

xp_stopmail system procedure

Function

Closes a MAPI e-mail session.

Syntax

[*variable* =] [**CALL**] **xp_stopmail ()**

Permissions

Not supported on NetWare.

Remarks

xp_stopmail is a system stored procedure that ends an e-mail session.

Return codes

☞ For a list of return codes, see "MAPI return codes" on page 854.

xp_stopsmtp system procedure

Function

Closes an SMTP e-mail session.

Syntax

[*variable* =] [**CALL**] **xp_stopsmtp ()**

Permissions

Not supported on NetWare

Remarks

xp_stopsmtp is a system stored procedure that ends an e-mail session.

Return codes

☞ For a list of return codes, see "SMTP return codes" on page 855.

Return codes for MAPI and SMTP stored procedures

Extended stored procedures for MAPI and SMTP use the following return codes.

MAPI return codes

Return code	**Meaning**
0	Success.
2	xp_startmail failed.

Return code	Meaning
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.
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).

Return code	Meaning
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.
553	Requested action not taken: mailbox name not allowed. (For example, the mailbox syntax is incorrect).
554	Transaction failed.

Other system extended stored procedures

Additional system extended stored procedures allow you to execute system commands. For example, you can use these stored procedures for file I/O and string operations.

xp_cmdshell system procedure

Function Carries out an operating system command from a procedure.

Syntax [*variable* = **CALL**] **xp_cmdshell (** *string* [, **'no_output'**] **)**

Arguments

- ***string*** This char(254) parameter specifies a system command.
- **'no_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 default behavior is to display output. If the second parameter is the string **'no_output'**, no output is displayed.

The second parameter affects only console applications on Windows NT/2000/XP 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.

Permissions

None

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_msver system procedure

Function

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 Adaptive Server Anywhere).
ProductVersion	The version number, followed by the build number. The format is as follows: `9.0.0 (1200)`
CompanyName	Returns the following string: `iAnywhere Solutions, Inc.`

Argument	Description
FileDescription	Returns the name of the product, followed by the name of the operating system.
LegalCopyright	Returns a copyright string for the software.
LegalTrade-marks	Returns trademark information for the software.

Remarks

xp_msver returns product, company, version, and other information.

Permissions

None

See also

♦ “System functions” on page 102

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 Adaptive Server Anywhere Windows NT

xp_read_file system procedure

Function

Returns the contents of a file as a LONG BINARY variable.

Syntax

[*variable* = **CALL**] **xp_read_file (** *filename* **)**

Arguments

♦ ***filename*** This long varchar parameter specifies a file name.

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 860

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_sprintf system procedure

Function	Builds up a string from a format string and a set of input strings
Syntax	[*variable* = **CALL**] **xp_sprintf (** *out-string*, *format-string* [, *input-string*] **)**

Arguments

- ♦ ***out-string*** This char(254) OUT parameter contains the result string.
- ♦ ***format-string*** This char(254) parameter specifies the format string.
- ♦ ***input-string*** This char(254) parameter specifies the input string.

Remarks

xp_sprintf builds up a string from a format string and a set of input strings. The format-string can contain up to fifty string placeholders (*%s*). These placeholders are filled in by the *input-string* arguments.

All arguments must be strings of less than 254 characters.

Permissions

None

Example

The following statements put the string *Hello World!* into the variable **mystring**.

```
CREATE VARIABLE mystring CHAR(254) ;
call xp_sprintf( mystring, ’Hello %s’, ’World!’ );
```

xp_scanf system procedure

Function	Extracts substrings from an input string and a format string.
Syntax	[*variable* = **CALL**] **xp_scanf (** *in-string*, *format-string* [, *output-string*] **)**

Arguments

- ♦ ***in-string*** This char(254) parameter specifies the input string.
- ♦ ***format-string*** This char(254) parameter specifies the format string.
- ♦ ***output-string*** This char(254) OUT parameter contains the result string.

Remarks

xp_scanf extracts substrings from an input string and a format string. The *format-string* can contain up to fifty string placeholders (*%s*). The values of these placeholders are placed in the *output-string* variables.

All arguments must be strings of less than 254 characters.

Permissions

None

Example

The following statements put the string `World!` into the variable **mystring**.

```
CREATE VARIABLE mystring CHAR(254) ;
call xp_scanf( 'Hello World!', 'Hello %s', mystring );
```

xp_write_file system procedure

Function

Writes data to a file from a SQL statement.

Syntax

[*variable* = **CALL**] **xp_write_file (** *filename*, *file_contents* **)**

Arguments

- ***filename*** This long varchar parameter specifies the file name.
- ***file_contents*** This long binary parameter specifies 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* 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 858

Example

Consider a table t1 that has the following columns:

- **filename** A filename relative to the server.
- **picture** A LONG BINARY column holding an image.

The following statement unloads the pictures into the named files:

```
SELECT xp_write_file( filename, picture)
FROM t1
```

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. Adaptive Server Anywhere has implemented support for some of these procedures.

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.

Adaptive Server Enterprise system procedures

The following list describes the Adaptive Server Enterprise system procedures that are provided in Adaptive Server 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 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	Description
sp_addgroup *group-name*	Adds a group to a database
sp_addlogin *userid*, *password*[, *defdb* [, *deflanguage* [, *fullname*]]]	Adds a new login ID to a database
sp_addmessage *message-num*, *message_text* [, *language*]	Adds a user-defined message to SYSUSERMESSAGES, for use by stored procedure PRINT and RAISERROR calls

System procedure	Description
sp_addtype *typename*, *data-type* [, "*identity*" | *nulltype*]	Creates a user-defined data type
sp_adduser *login_name* [, *name_in_db* [, *grpname*]]	Adds a new user ID to a database
sp_changegroup *new-group-name*, *userid*	Changes a user's group or adds a user to a group
sp_dboption [*dbname*, *optname*, {*true* | *false*}]	Displays or changes a database option
sp_dropgroup *group-name*	Drops a group from a database
sp_droplogin *userid*	Drops a login ID from a database
sp_dropmessage *message-number* [, *language*]	Drops a user-defined message
sp_droptype *typename*	Drops a user-defined data type
sp_dropuser *userid*	Drops a user ID from a database
sp_getmessage *message-num*, *@msg-var output* [, *language*]	Retrieves a stored message string from SYSUSER-MESSAGES, for PRINT and RAISERROR statements.
sp_helptext *object-name*	Displays the text of a system procedure, trigger, or view
sp_password *caller_passwd*, *new_passwd* [, *userid*]	Adds or changes a password for a user ID

Adaptive Server Enterprise catalog procedures

Adaptive Server Anywhere implements a subset of the Adaptive Server Enterprise catalog procedures. The implemented catalog procedures are described in the following table.

Catalog procedure	Description
sp_column_privileges	Unsupported
sp_columns *table-name* [, *table-owner*] [, *table-qualifier*] [, *column-name*]	Returns the data types of the specified columns
sp_databases	Unsupported
sp_datatype_info	Unsupported
sp_fkeys *pktable_name* [, *pktable-owner*][, *pktable-qualifier*] [, *fktable-name*] [, *fktable_owner*] [, *fktable-qualifier*]	Returns foreign key information about the specified table
sp_pkeys *table-name* [, *table_owner*] [, *table_qualifier*]	Returns primary key information about the specified table
sp_server_info	Unsupported
sp_special_columns *table_name* [, *table-owner*] [, *table-qualifier*] [, *col-type*]	Returns the optimal set of columns that uniquely identify a row in the specified table
sp_sproc_columns *proc-name* [, *proc_owner*] [, *proc-qualifier*] [, *column-name*]	Returns information about a stored procedure's input and return parameters
sp_stored_procedures [*sp-name*] [, *sp-owner*] [, *sp-qualifier*]	Returns information about one or more stored procedures
sp_statistics [*table_name*] [, *table_owner*] [, *table_qualities*] [, *index_name*] [, *is_unique*]	Returns information about tables and their indexes

Catalog procedure	Description
`sp_tables` *`table-name`* `[,` *`table-owner`*`] [,` *`table-qualifier`*`] [,` *`table-type`*`]`	Returns a list of objects that can appear in a FROM clause for the specified table

Index