CHAPTER 6

SQL Language Elements

About this chapter

This chapter presents detailed descriptions of the language elements and conventions of SQL.

Contents

Statement elements

The following elements are found in the syntax of many SQL statements.

- **column-name** An identifier that represents the name of a column.
- **condition** An expression that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 194.
- **connection-name** An identifier or a string representing the name of an active connection.
- **owner** An identifier that represents a user ID.
- **data-type** A storage data type, as described in "SQL Data Types" on page 219.
- **expression** An expression, as described in "Expressions" on page 183.
- **filename** A string containing a filename.
- **host-variable** A C language variable, declared as a host variable preceded by a colon.
- **identifier** Any string of alphabetic characters, or digits. The collation sequence of the database dictates which characters are considered alphabetic or digit characters. The underscore character (_), at sign (@), number sign (#), and dollar sign ($) are considered alphabetic characters. The first character must be an alphabetic character.

  Alternatively, any string of characters can be used as an identifier by enclosing it in quotation marks ("double quotes").

  A quotation mark inside the identifier is represented by two quotation marks in a row. Identifiers are truncated to 128 characters. The following are all valid identifiers.

  ```
  Surname

  "Surname"

  SomeBigName

  some_big_name

  "Client Number"

  "With a quotation "" mark"
  ```

  Some words that adhere to the above rules are reserved because they play an important role in the syntax of the SQL language. If you choose to use one of them as an identifier, you must enclose it in double quotes.

☞ For a complete list of the reserved words, see "Alphabetical list of keywords" on page 215.

- **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
  ```

- **role-name** An identifier representing the role name of a foreign key.
- **search-condition** A condition that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 194.
- **string** Any sequence of characters enclosed in apostrophes ('single quotes').

  To represent an apostrophe inside the string, use two apostrophes in a row. To represent a new line character, use a backslash followed by n (\n). To represent a backslash character, use two backslashes in a row (\\).

  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). The following are valid strings:

  ```
  'This is a string.'
  'John''s database'
  '\x00\x01\x02\x03'
  ```

  For compatibility with Adaptive Server Enterprise, you can set the QUOTED_IDENTIFIER database option to OFF, and then you can also use double quotes to mark the beginning and end of strings. The option is set to ON by default.
- **savepoint-name** An identifier that represents the name of a savepoint.
- **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. See "FROM clause" on page 476.
- **table-name** An identifier that represents the name of a table.
- **userid** An identifier that represents a user name.
- **variable** An identifier that represents a variable name.

Expressions

Syntax

```
expression:
    case-expression
   | constant
   | [correlation-name .] column-name [ java-ref ]
   | - expression
   | expression operator expression
   | ( expression )
   | function-name ( expression, ... )
   | if-expression
   | [ java-package-name.] java-class-name java-ref
   | ( subquery )
   | variable-name [ java-ref ]
```

Parameters

```
case-expression:
   {  CASE search-condition
   ...   WHEN expression
   ...   THEN,...
   ...   [ ELSE expression ]
      END
   |  CASE
   ...   WHEN search-condition
   ...   THEN expression,...
   ...   [ ELSE expression ]
      END
   }

constant:
    integer  | number  | 'string'  | special-constant  | host-variable

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

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

java-ref:
    .field-name [ java-ref ]
   | >> field-name [ java-ref ]
   | .method-name ( [expression,...] ) [ java-ref ]
   | >> method-name ( [expression,...] ) [ java-ref ]

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

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

"Search conditions" on page 194
"SQL Data Types" on page 219
"SQL Functions" on page 267
"SQL variables" on page 203

Description

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

For information on functions, see "SQL Functions" on page 267. For information on variables, see "SQL variables" on page 203.

Compatibility

- The IF condition is not supported in Adaptive Server Enterprise.
- Java expressions are not currently 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 strings. String constants are enclosed in apostrophes ('single quotes'). An apostrophe is represented inside the string by two apostrophes in a row.

Special constants

There are several special constants:

- **CURRENT DATE** The current year, month and day represented in the DATE data type.
- **CURRENT TIME** The current hour, minute, second and fraction of a second represented in the TIME data type. Although the fraction of a second is stored to 6 decimal places, the current time is limited by the accuracy of the system clock.
- **CURRENT TIMESTAMP** Combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value containing the year, month, day, hour, minute, second and fraction of a second. Like CURRENT TIME, the accuracy of the fraction of a second is limited by the system clock.
- **NULL** The NULL value (see "NULL value" on page 213).

- **SQLCODE** Current SQLCODE value (see "Database Error Messages" on page 581).
- **SQLSTATE** Current SQLSTATE value (see "Database Error Messages" on page 581).
- **CURRENT USER** A string containing the user ID of the current connection.

  On UPDATE, columns with a default of CURRENT USER automatically update to reflect the current connection.
- **CURRENT PUBLISHER** A string containing the publisher user ID of the database for SQL Remote replications.
- **LAST USER** For 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 altered, it is changed to the name of the current user. In this way, the LAST USER default indicates the user ID of the user who last modified the row.

  When combined with the CURRENT 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.

In Embedded SQL, a host variable can also be used in an expression wherever a constant is allowed.

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 476.) 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"
```

See "Statement elements" on page 180 for a complete description of identifiers.

Functions in expressions

See "SQL Functions" on page 267 for a description of SQL functions.

Subqueries in expressions

A subquery is a SELECT statement enclosed in parentheses. The SELECT statement must contain one and only one select list item. Usually, the subquery is allowed to return only one row. See "Search conditions" on page 194 for other uses of subqueries. 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.

SQL Operators

This section describes arithmetic, string, and bitwise operators. For information on comparison operators, see the section "Search conditions" on page 194.

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.

Arithmetic operators

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

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

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

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

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

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.

String operators

expression || expression String concatenation (two vertical bars). If either string is the NULL value, 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.

Standards and compatibility

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

Bitwise operators

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

Operator	Description
&	and
|	or
^	exclusive or
~	not

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.

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 concatenation** When using the + concatenation operator in Adaptive Server Anywhere, you should explicitly set the operands to strings 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.

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

Operator precedence

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.

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 NULL, the IF expression returns NULL.

For more information about TRUE, FALSE and UNKNOWN conditions, see "NULL value" on page 213, and "Search conditions" on page 194.

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 489.

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.

Java expressions

The following kinds of Java expressions can be used as SQL expressions:

♦ **Java fields** Any field of an installed Java class can be invoked wherever an expression is required. The data type of the expression is converted from the Java field data type according to the table in "Java to-SQL-data type conversion" on page 256. Both instance fields and class fields can be used as expressions.

♦ **Java methods** Any method of an installed Java class can be invoked wherever an expression is required. The data type of the expression is converted from the return type of the Java method according to the table in "Java to-SQL-data type conversion" on page 256. Both instance fields and class fields can be used as expressions.

♦ **Java objects** The NEW operator is an extension to the SQL language that allows it to better assimilate Java syntax.

  The NEW SQL operator performs the same operation as the **new** keyword in Java code: invoke a constructor method of a Java class. The data type of the NEW expression is a Java class, specifically the Java class that is being constructed.

  The following expression invokes the constructor method of the String class, a member of the `java.lang` package.

  ```
  NEW java.lang.String( 'This argument is optional' )
  ```

  This expression returns a reference to the newly-created String object, which can be passed to a variable or column of type **java.lang.String**.

  The method constructor that is being invoked determines the number and type of arguments.

  The class whose constructor method is invoked must first be installed to the database.

For more information on class and instance fields and methods, see "A Java seminar" on page 439 of the book *Adaptive Server Anywhere User's Guide*.

Referencing fields and methods

When referencing a Java field or method from within Java code, you use the dot (.) operator. For example, to invoke the `getConnection` method of the `DriverManager` class you use the following:

```
conn = DriverManager.getConnection( temp.toString() ,
_props )
```

There are two ways of referencing Java fields or methods from within SQL statements. You can use either the dot operator or the >> operator.

The dot operator has the advantage that it looks like Java code, but has the disadvantage that in SQL the dot is also used to indicate the owner, table, and column hierarchy, so this could be confusing to read.

Using the dot operator, a `name` method of an object named `Employee` is invoked from SQL as follows:

```
select Employee.name ...
```

The same expression could refer to a **name** column of an **Employee** table.

The >> operator is unambiguous, but does not look like what Java programmers may expect.

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 "Alphabetical list of keywords" on page 215.

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 DBMS's.

```
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, a JOIN clause, 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*
 | *condition* **AND** *condition*
 | *condition* **OR** *condition*
 | (*condition*)
 | (*condition* , *estimate*)
 | *condition* **IS** [**NOT**] { **TRUE** | **FALSE** | **UNKNOWN** }

Parameters

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

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

"Expressions" on page 183

Description

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.

SQL conditions do not follow boolean logic, where conditions are either true or false. 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 showing how logical operators combine in three-valued logic, see the section "Three-valued logic" on page 201.

Rows satisfy a search condition if and only if the result of the condition is TRUE. Rows for which the condition is UNKNOWN do not satisfy the search condition. For more information about NULL, see "NULL value" on page 213.

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 196.

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

Comparison conditions

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

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

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.
- **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 when creating the database.

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 conditions" on page 195) 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 is 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.

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:

> *expr* >= *start-expr* **AND** *expr* <= *end-expr*

Compatibility

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

LIKE conditions

The syntax for LIKE conditions is as follows:

> ... *expression* [**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 wild cards. The wild cards are:

Wild card	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 "Database Collations and International Languages" on page 289 of the book *Adaptive Server Anywhere User's Guide*.

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.

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 wild card 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 wild cards 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.*

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 )
      |       expression [ NOT ] IN ( expression )
      |       expression [ NOT ] IN (value-expr1 , value-expr2 [, value-
      expr3 ] ... )
```

Without the NOT keyword, the IN conditions is TRUE if *expression* equals any of the listed values, UNKNOWN if *expression* is the NULL value, and FALSE otherwise. The NOT keyword reverses the meaning of the condition, but leaves UNKNOWN unchanged.

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.

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.

Compatibility

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

Conditions with 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.

Compatibility

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

NOT conditions

The syntax for NOT conditions is as follows:

... **NOT** *condition1*

The NOT condition is TRUE if *condition1* is FALSE, FALSE if *condition1* is TRUE, and UNKNOWN if *condition1* is UNKNOWN.

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.

Compatibility

- Truth-valued conditions are supported by Adaptive Server Anywhere only.

Three-valued logic

The following tables show 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

User-supplied estimates

The Adaptive Server Anywhere query optimizer uses educated guesses to help decide the best strategy for executing a query. For each table in a potential execution plan, the optimizer must estimate the number of rows that will be part of the results. If you know that a condition has a success rate that differs from the optimizer rule, you can tell the database this information by using an estimate in the search condition.

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

Examples

♦ The following query provides an estimate that one percent of the **ship_date** values will be later than 1994/06/30:

```
SELECT  ship_date
FROM  sales_order_items
WHERE ( ship_date > '1994/06/30', 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.

Compatibility

♦ Adaptive Server Enterprise does not support explicit estimates.

SQL 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.

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. 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 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 ;
END
```

Running this batch from Interactive SQL gives the message local_var = 10 on the server window.

- 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 ;
     MESSAGE 'local_var = ', local_var ;
  END;
  MESSAGE 'local_var = ', local_var ;
  ```

- 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 ;
  END
  ```

  Running this batch from Interactive SQL gives the message local_var = 10 on the server window.

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" on page 810 of the book *Adaptive Server Anywhere User's Guide*.

Connection-level variables

Connection-level variables are declared with the CREATE VARIABLE statement. The CREATE VARIABLE statement can be used anywhere except inside compound statements. Connection-level variables can be passed as parameters to procedures.

The syntax for the CREATE VARIABLE statement is as follows:

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

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

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

```
DROP VARIABLE con_var
```

Example

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

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

  Running this batch from Interactive SQL gives the message local_var = 10 on the server window.

Compatibility

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

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 value 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 (CURRENT DATE, CURRENT TIME, USER, SQLSTATE and so on) 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
            NUMERIC ( 5, 2 ) )
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
@@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.
@@identity	Last value inserted into an IDENTITY column by an INSERT or SELECT INTO statement. @@identity 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.
@@isolation	Current isolation level. @@isolation takes the value of the active level.
@@procid	Stored procedure ID of the currently executing procedure.
@@rowcount	Number of rows affected by the last statement. @@rowcount is set to zero by any statement which does not affect rows, such as an IF 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.
@@servername	Name of the current database server.
@@sqlstatus	Contains status information resulting from the last fetch statement.
@@version	Version number of the current version of Adaptive Server Anywhere.

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
@@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. @@identity 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
@@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

Global variable	Returns
	Server Anywhere, returns 0
@@isolation	Current isolation level of the connection. In Adaptive Server Enterprise, @@isolation takes the value of the active level
@@langid	In Adaptive Server Enterprise, defines the local language ID of the language currently in use. In Adaptive Server Anywhere, returns 0
@@language	In Adaptive Server Enterprise, defines the name of the language currently in use. In Adaptive Server Anywhere, returns "English"
@@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
@@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. @@rowcount is set to zero by any command which does not return rows, such as an IF statement. With cursors, @@rowcount represents the cumulative number of rows returned from the cursor result set to the client, up to the last

Global variable	Returns
	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
@@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

Global variable	Returns
@@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

SQL 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.

Transact-SQL 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 183 "Search conditions" on page 194

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. Note that these 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 the NULL value.

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 194 for a description of this clause.

In the following examples, the column **Salary** contains the NULL value.

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

The NULL value also has an interesting property when used in numeric expressions. The result of *any* numeric expression involving the NULL value is the NULL value. This means that if the NULL value is added to a number, the result is the NULL value—not a number. If you want the NULL value to be treated as 0, you must use the **ISNULL(** *expression*, **0)** function (see "SQL Functions" on page 267).

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 194 for a description of the effect of three-valued logic when combining search conditions.

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' )
```

Alphabetical list of keywords

To use a keyword (also called a reserved word) as an identifier, you must enclose it in double quotes when referencing it in a SQL statement. Many, but not all, of the words that appear in SQL statements are keywords. 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, 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.

List of reserved words

add	all	alter	and
any	as	asc	backup
begin	between	bigint	binary
bit	bottom	break	by
call	cascade	case	cast
char	char_convert	character	check
checkpoint	close	comment	commit
connect	constraint	continue	convert
create	cross	current	cursor
date	dbspace	deallocate	dec
decimal	declare	default	delete
desc	disable	distinct	do
double	drop	dynamic	else
elseif	enable	encrypted	end
endif	escape	exception	exec
execute	existing	exists	externlogin
fetch	first	float	for

List of reserved words

foreign	forward	from	full
goto	grant	group	having
holdlock	identified	if	in
index	inner	inout	insensitive
insert	install	instead	int
integer	integrated	into	iq
is	isolation	join	key
left	like	lock	login
long	match	membership	message
mode	modify	natural	new
no	noholdlock	not	notify
null	numeric	of	off
on	open	option	options
or	order	others	out
outer	passthrough	precision	prepare
primary	print	privileges	proc
procedure	publication	raiserror	readtext
real	reference	references	release
remote	remove	rename	resource
restore	restrict	return	revoke
right	rollback	save	savepoint
schedule	scroll	select	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	union	unique	unknown

List of reserved words

unsigned	update	user	using
validate	values	varbinary	varchar
variable	varying	view	when
where	while	with	work
writetext			