CHAPTER 7

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 are stored separate from the row containing the long string value.

There are several functions (see "SQL Functions" on page 267) 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 and precision 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 32,767.

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 ASE is 255.
- **Other database systems** In many other database management systems, unlike Adaptive Server Anywhere,CHAR data types require *max-length* bytes of storage, regardless of the length of the actual string.

See also
"CHARACTER VARYING data type" on page 221
"LONG VARCHAR data type" on page 222

CHARACTER VARYING data type [Character]

Function

Syntax Same as CHAR.

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

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 32,767.

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" on page 221
"LONG VARCHAR data type" on page 222

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 gigabytes).

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" on page 221
"CHARACTER VARYING data type" on page 221

TEXT data type [Character]

Function

This is a user-defined data type. 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" on page 222

Numeric data types

Function

For storing numerical data.

Notes

- The NUMERIC and DECIMAL data types, and the various kinds of INTEGER data type, 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 accurate guaranteed accurate to the least significant digit specified after an arithmetic operation.*

- TINYINT columns should not be fetched into Embedded SQL 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.

- 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

- In Embedded SQL, TINYINT columns should be fetched into 2-byte or 4-byte integer columns. Also, to send a TINYINT value to a database, the C variable should be an integer.

- 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 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.

BIGINT data type [Numeric]

Function

A signed 64-bit integer.

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 from **(-2e63 + 1)** to **(+2e63 -1)**.

The range for unsigned BIGINT values is from 0 to **(2e64 - 1)**.

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" on page 227
"TINYINT data type" on page 229
"SMALLINT data type" on page 229

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 )
```

where **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" on page 168 and "SCALE option" on page 172.

Standards and compatibility

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

See also

"FLOAT data type" on page 226
"REAL data type" on page 228
"DOUBLE data type" on page 226

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.

It is an approximate numeric data type; subject to roundoff 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 range of values 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" on page 226
"REAL data type" on page 228
"DECIMAL data type" on page 225

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 roundoff 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 places after the decimal.

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" on page 156.

See also

"DECIMAL data type" on page 225
"REAL data type" on page 228
"DOUBLE data type" on page 226

INT or INTEGER data type [Numeric]

Function

An integer value 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 from **(-2e31 + 1)** to **(+2e31 -1)**.

The range for unsigned integers is from **0** to **(2e32 - 1)**.

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" on page 225
"TINYINT data type" on page 229

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 storage required for a decimal number can be estimated as:

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

where **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" on page 168 and "SCALE option" on page 172.

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" on page 226
"REAL data type" on page 228
"DOUBLE data type" on page 226

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 1.175495e-38 to 3.402823e+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 value 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 from -32768 to + 32767.

The range for unsigned SMALLINT values is from zero 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" on page 227
"TINYINT data type" on page 229
"BIGINT data type" on page 225

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 from zero to 255.

Standards and compatibility

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

See also

"BIGINT data type" on page 225
"TINYINT data type" on page 229
"SMALLINT data type" on page 229

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 NUMERIC(19,4), allowing NULL.

Standards and compatibility

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

See also "SMALLMONEY data type" on page 231

SMALLMONEY data type [Money]

Function The 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 user-defined data type, as NUMERIC(10,4), allowing NULL.

Standards and compatibility

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

See also "MONEY data type" on page 231

Bit data type

Function

For storing Boolean values.

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.

Standards and compatibility

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

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 553). 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 Oct 11 1912, Nov 12 1910, or Nov 10 1912. The year, month, and day of a date string should be separated by some character (for example /, -, or space) and appear in the order specified by the DATE_ORDER option. The year can be supplied as either 2 or 4 digits, with 2 digit years defaulting to the 20th century. 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

- As the year 2000 approaches it might be a good idea to specify the year using the four-digit format. This will ensure the integrity of the data into the next millenium.
- With an appropriate setting of DATE_ORDER, the following strings are all valid dates:

  ```
  92-05-23 21:35
  92/5/23
  1992/05/23
  May 23 1992
  23-May-1992
  Tuesday May 23, 1992 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 1992 will be the date 1992-05-01 00:00)
- **hour, minute, second, fraction** 0

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 553.

For information on functions that deals with dates and times, see "Date and time functions" on page 269. 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.

Date and time comparisons

The DATE data type also contains a time. 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 always involve the times as well. A database date value of '1992-05-23 10:00' will not be equal to the constant '1992-05-23'. 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') = '1992/05/23'
```

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.

DATE data type [Date and Time]

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

Syntax **DATETIME**

Usage The year can be from the year 0001 to 9999. For historical reasons, a DATE column can also contain an hour and minute, but the TIMESTAMP data type is now recommended for anything with hours and minutes. A DATE value requires 4 bytes of storage.

Standards and compatibility

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

See also "DATETIME data type" on page 236
"SMALLDATETIME data type" on page 236

DATETIME data type [Date and Time]

Function A user-defined data type, implemented as TIMESTAMP.

Syntax **SMALLDATETIME**

Usage DATETIME is provided primarily for compatibility with Adaptive Server Enterprise.

Standards and compatibility

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

See also "DATE data type" on page 236
"SMALLDATETIME data type" on page 236

SMALLDATETIME data type [Date and Time]

Function A user-defined data type, 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.

See also "DATE data type" on page 236
"DATETIME data type" on page 236

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** Not supported by Adaptive Server Enterprise.

See also "TIMESTAMP data type" on page 237

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" on page 237

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 32,767. The BINARY data type is identical to the CHAR data type except when used in comparisons. BINARY values will be compared exactly while CHAR values are compared without respect to upper/lower case unless the database is defined as case-sensitive or accented characters.

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" on page 239
"VARBINARY data type" on page 240

LONG BINARY data type [BINARY]

Function Arbitrary length binary data. The maximum size is limited by the maximum size of the database file (currently 2 gigabytes).

Syntax **LONG BINARY**

Usage The maximum size is limited by the maximum size of the database file (currently 2 gigabytes).

Standards and compatibility

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

See also

"BINARY data type" on page 239
"VARBINARY data type" on page 240

IMAGE data type [BINARY]

Function

LONG BINARY data allowing NULL.

Syntax

IMAGE

Usage

IMAGE is implemented in Adaptive Server Anywhere as a user-defined data type, 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.

VARBINARY data type [BINARY]

Function

Identical to BINARY.

Syntax

VARBINARY [(*n*)]

Usage

Variable length binary strings. The default value for *n* is one.

Standards and compatibility

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

See also

"BINARY data type" on page 239
"LONG BINARY data type" on page 239

User-defined data types

Function

User-defined data types are aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions.

User-defined data types, also called **domains**, 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. This encourages consistency throughout the database.

Simple user-defined data types

User-defined data types are created using the CREATE DOMAIN statement For full description of the syntax, see "CREATE DOMAIN statement" on page 391.

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
)
```

User-defined data types 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.

Constraints and defaults with user-defined data types

Many of the attributes associated with columns, such as allowing NULL values, having a DEFAULT value, and so on, can be built into a user-defined data type. 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, user-defined data types 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 user-defined data type, or you can use the CREATE DOMAIN statement. In Adaptive Server Enterprise, you must use **sp_addtype**.

Java class data types

Function

Any Java class that is installed into a database can be used as a SQL data type. Java class data types provide abstract data types for use within the database.

Built-in and user-defined Java classes

Java classes in the database fall into one of the following categories:

- **Built-in classes** A subset of the Java API is installed into all Java-enabled databases.
- **User-defined classes** Users with DBA permission can install compiled Java classes into a database.

Case sensitivity of Java class data types

Java identifiers, including data types, are case sensitive: the Java `int` data type cannot be written as `INT`. SQL identifiers, including data types, are case insensitive. The **int** data type can also be written as **Int** or any other combination of upper and lower case characters.

When a Java class is used as a SQL data type, the data type is always case sensitive. This is an exception to the rule for SQL identifier case insensitivity.

Example

If you install a class named Demo, the following statements are *not* equivalent:

```
CREATE TABLE t1 (
   id INT PRIMARY KEY
   demo_column Demo )

CREATE TABLE t1 (
   id INT PRIMARY KEY
   demo_column DEMO )
```

Built-in Java classes

This section lists the built-in classes available for use as SQL data types in a Java-enabled database.

java.beans classes

The following table lists the supported classes, interfaces, and exceptions from *java.beans*.

Class	Interface	Exception
BeanDescriptor	BeanInfo	IntrospectionException
Beans	Customizer	PropertyVetoException
EventSetDescriptor	PropertyChangeListener	
FeatureDescriptor	PropertyEditor	
IndexedPropertyDescriptor	VetoableChangeListener	
Introspector	Visibility	
MethodDescriptor		
ParameterDescriptor		
PropertyChangeEvent		
PropertyChangeSupport		
PropertyDescriptor		
PropertyEditorManager		
PropertyEditorSupport		
SimpleBeanInfo		
VetoableChangeSupport		

java.lang classes

The following table lists the supported classes, interfaces, and exceptions from *java.lang*.

Class	Interface	Exception
Boolean	Cloneable	ArithmeticException
Byte	Runnable	ArrayIndexOutOfBoundsException
Character		ArrayStoreException
Class		ClassCastException
ClassLoader		ClassNotFoundException
Compiler		CloneNotSupportedException
Double		Exception
Float		IllegalAccessException
Integer		IllegalArgumentException
Long		IllegalMonitorStateException

Class	Interface	Exception
Math		IllegalStateException
Number		IllegalThreadStateException
Object		IndexOutOfBoundsException
Process		InstantiationException
Runtime		InterruptedException
(not supported: SecurityManager)		NegativeArraySizeException
Short		NoSuchFieldException
String		NoSuchMethodException
StringBuffer		NullPointerException
System (not supported: Thread, ThreadGroup)		NumberFormatException
Throwable		RuntimeException
Void		SecurityException
		StringIndexOutOfBoundsException

java.lang.reflect classes

The following table lists the supported classes, interfaces, and exceptions from *java.lang.reflect*.

Class	Interface	Exception
Array	Member	InvocationTargetException
Constructor		
Field		
Method		
Modifier		

java.math classes

The following table lists the supported classes from *java.math*.

Class	Interface	Exception
BigDecimal		
BigInteger		

java.sql classes

The following table lists the supported classes, interfaces, and exceptions from *java.sql*.

Class	Interface	Exception
Date	CallableStatement	DataTruncation
DriverManager	Connection	SQLException
DriverPropertyInfo	DatabaseMetaData	SQLWarning
Time	Driver	
Timestamp	PreparedStatement	
Types	ResultSet	
	ResultSetMetaData	
	Statement	

java.text classes

The following table lists the supported classes, interfaces, and exceptions from *java.text*.

Class	Interface	Exception
BreakIterator	CharacterIterator	ParseException
ChoiceFormat		
CollationElementIterator		
CollationKey		
Collator		
DateFormat		
DateFormatSymbols		
DecimalFormat		
DecimalFormatSymbols		

Class	Interface	Exception
FieldPosition		
Format		
MessageFormat		
NumberFormat		
ParsePosition		
RuleBasedCollator		
SimpleDateFormat		
StringCharacterIterator		

java.util classes

The following table lists the supported classes, interfaces, and exceptions from *java.util*.

Class	Interface	Exception
BitSet	Enumeration	EmptyStackException
Calendar	EventListener	MissingResourceException
Date	Observer	NoSuchElementException
Dictionary		TooManyListenersException
EventObject		
GregorianCalendar		
Hashtable		
ListResourceBundle		
Locale		
Observable		
Properties		
PropertyResourceBundle		
Random		
ResourceBundle		

Class	Interface	Exception
SimpleTimeZone		
Stack		
StringTokenizer		
TimeZone		
Vector		

java.util.zip classes (Compression utilities)

The following table lists the supported classes, interfaces, and exceptions from *java.util*.

Class	Interface	Exception
Adler32	Checksum	DataFormatException
CRC32		ZipException
CheckedInputStream		
CheckedOutputStream		
Deflater		
DeflaterOutputStream		
GZIPInputStream		
GZIPOutputStream		
Inflater		
InflaterInputStream		
ZipEntry		
ZipFile		
ZipInputStream		
ZipOutputStream		

java.io classes

The following table lists the supported classes, interfaces, and exceptions from *java.io*.

Class	Interface	Exception
BufferedInputStream	DataInput	CharConversionException
BufferedOutputStream	DataOutput	EOFException
BufferedReader	Externalizable	FileNotFoundException
BufferedWriter	FilenameFilter	IOException
ByteArrayInputStream	ObjectInput	InterruptedIOException
ByteArrayOutputStream	ObjectInputValidation	InvalidClassException
CharArrayReader	ObjectOutput	InvalidObjectException
CharArrayWriter	Serializable	NotActiveException
DataInputStream		NotSerializableException
DataOutputStream		ObjectStreamException
File		OptionalDataException
FileDescriptor		StreamCorruptedException
FileInputStream		SyncFailedException
FileOutputStream		UTFDataFormatException
FileReader		UnsupportedEncodingException
FileWriter		WriteAbortedException
FilterInputStream		
FilterOutputStream		
FilterReader		
FilterWriter		
InputStream		
InputStreamReader		
LineNumberInputStream		
LineNumberReader		
ObjectInputStream		
ObjectOutputStream		
ObjectStreamClass		
OutputStream		
OutputStreamWriter		
PipedInputStream		

Class	Interface	Exception
PipedOutputStream		
PipedReader		
PipedWriter		
PrintStream		
PrintWriter		
PushbackInputStream		
PushbackReader		
RandomAccessFile		
Reader		
SequenceInputStream		
StreamTokenizer		
StringBufferInputStream		
StringReader		
StringWriter		
Writer		

Unsupported packages and classes

The following packages are not supported in the Sybase VM:

- java.applet
- java.awt
- sun.applet
- sun.audio
- sun.awt
- sun.beans.editors
- sun.beans.infos
- sun.jdbc.odbc
- sun.misc
- sun.net
- sun.rmi

- sun.security
- sun.tools
- sunw.io
- sunw.util

The following classes are only partially supported. They may have some unsupported native methods

- java.lang.ClassLoader
- java.lang.Compiler
- java.lang.Runtime (exec/load/loadlibrary)
- java.lang.SecurityManager
- java.lang.Thread
- java.io.File
- java.io.FileDescriptor
- java.io.FileInputStream
- java.io.FileOutputStream
- java.io.ObjectInputStream
- java.io.ObjectOutputStream
- java.io.ObjectStreamClass
- java.io.PrintStream
- java.io.RandomAccessFile
- java.math.BigInteger
- java.net.PlainDatagramSocketImpl
- java.util.zip.Adler32
- java.util.zip.CRC32
- java.util.zip.Deflater
- java.util.zip.Inflater

User-defined Java classes

Users with DBA permissions can install Java classes into a database. Any class installed into the database becomes available as a data type.

Preparing classes using the JDK

The Java Development Kit (JDK) provides the tools necessary for preparing classes for installation into a database.

❖ **To prepare a class for installation, using the JDK:**

1. Using a text editor, write a Java class, outside the database, and store it as a Java source code file (typically with an extension of *.java*).

   For example, you may create a file named *MyFirstClass.java*.

2. Using the *javac* compiler, compile the Java class to produce a Java class file (with *.class* extension).

   For example, to compile the file *MyFirstClass.java*, type the following at a command prompt:

   ```
   javac MyFirstClass.java
   ```

Installing a class

Once you have a compiled Java class file, you can install it into the database. You can do this conveniently using either Sybase Central or Interactive SQL.

❖ **To install a class, using Sybase Central:**

1. From Sybase Central, connect to the database as a user ID with DBA permissions.
2. Open the Java Objects folder, and double-click Add Java Class or Jar. Follow the instructions in the wizard to install the class.

❖ **To install a class, using Interactive SQL:**

1. From Interactive SQL, connect to the database as a user ID with DBA permissions.
2. Enter the following command to install the class:

   ```
   INSTALL JAVA NEW FROM filename
   ```

Using classes as data types

Creating tables using Java class data types

You can create a table with columns based on a Java class data type, just as you can with any other data type. For example, if **MyClass** is a Java class installed into the database, you can create a table using this class as follows:

```
CREATE TABLE mytable (
   id INT NOT NULL PRIMARY KEY,
   mycol MyClass )
```

In this statement, the **MyClass** data type is a SQL data type, and so is case-insensitive, even though Java is a case-sensitive language.

Inserting Java objects

You can insert a Java object into a table just as you would any other row, using the INSERT statement. Because each row is a separate instance of the class, you must use the NEW keyword to create an instance. For example,

```
INSERT INTO t2
VALUES ( 1, NEW MyClass() )
```

In this case, **MyClass()** is a Java class name, not a SQL data type, and so must be entered in the proper case.

In this example, **MyClass()** has no arguments, so each row is created using the default constructor. In general, you would supply arguments to a class to place distinct values in each row.

Using subclasses

If you install a class, say **`MySubClass`**, which is a subclass of **`MyClass`**, you can insert instances of **`MySubClass`** into a column of data type **MyClass**.

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 - Adaptive Server Anywhere tries to
convert the string to a number

'a' > 0 -- Adaptive Server Anywhere tries to convert 'a'
to a number
```

The CAST or CONVERT functions can be used to force type conversions. For information about the CAST and CONVERT functions, see "Data type conversion functions" on page 275.

The following functions can also be used to force type conversions (see "SQL Functions" on page 267).

- **DATE(value)** Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.
- **STRING(value)** Similar to CAST(value AS CHAR), except that string(NULL) is the empty string (''), while CAST(NULL AS CHAR) is the NULL value.
- **VALUE+0.0** Equivalent to CAST(value AS DECIMAL).

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 milliseconds 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.

Example

- Adaptive Server Enterprise would convert the values below as shown.

  `12:34:56.7` to `12:34:56.700`

  `12.34.56.78` to `12:34:56.780`

  `12:34:56.789` to `12:34:56.789`

  `12:34:56:7` to `12:34:56.007`

  `12.34.56:78` to `12:34:56.078`

  `12:34:56:789` to `12:34:56.789`

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

  `12:34:56.7` to `12:34:56.700`

  `12.34.56.78` to `12:34:56.780`

  `12:34:56.789` to `12:34:56.789`

  `12:34:56:7` to `12:34:56.700`

  `12.34.56:78` to `12:34:56.780`

  `12:34:56:789` to `12:34:56.789`

Java / SQL data type conversion

When a Java class field or method is invoked within a SQL statement, a Java data type is returned by the Java object. This must be converted into a SQL data type for use within the SQL statement, for example in comparisons.

Similarly, when a SQL statement is included in a JDBC class, it provides data of SQL data types, which must be converted to Java data types.

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 type	SQL type
java.sql.Timestamp	TIMESTAMP
java.lang.Double	DOUBLE
java.lang.Float	REAL
java.lang.Integer	INTEGER
java.lang.Long	INTEGER

SQL-to-Java data type conversion

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

Year 2000 compliance

The problem of handling dates, in particular year values beyond the year 2000, is 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 233.

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 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" on page 150.

For more information on unambiguous date formats, see "Unambiguous dates and times" on page 233.

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 233.

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" on page 17 of the book *Adaptive Server Anywhere Programming Interfaces Guide*.

Leap years

The year 2000 is also 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 Feb 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 Feb 29 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, Adaptive Server Anywhere 's 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 adding 1900 to two digit year strings and placing them in the 20th century.

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 0, thus Adaptive Server Anywhere converts the string into the date 1900-01-01. 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 = 25;
```

When NEAREST_CENTURY option is set to 25, 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 2000-01-01. 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 Jan 1, 2000 into a string representing the date Jan 1, 1900 even though no database error occurs.

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

Although the unambiguous date string 2000-01-01 is automatically and correctly converted by Adaptive Server Anywhere into a date value, 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 in the 20th century.

For more information on date and time functions, see "Date and time functions" on page 269.