Chapter 3. Language Elements

This chapter defines the basic syntax of SQL and language elements that are common to many SQL statements.

Characters

The basic symbols of keywords and operators in the SQL language are single-byte characters that are part of all IBM character sets. Characters of the language are classified as letters, digits, or special characters.

Characters

A *letter* is any of the 26 uppercase (A through Z) and 26 lowercase (a through z) letters plus the three characters ($, #, and @), which are included for compatibility with host database products (for example, in code page 850, $ is at X'24' # is at X'23', and @ is at X'40'). Letters also include the alphabetics from the extended character sets. Extended character sets contain additional alphabetic characters; for example, those with diacritical marks (́ is an example of a diacritical mark). The available characters depend on the code page in use.

A *digit* is any of the characters 0 through 9.

A *special character* is any of the characters listed below:

	blank	–	minus sign
"	quotation mark or double-quote	.	period
%	percent	/	slash
&	ampersand	:	colon
'	apostrophe or single quote	;	semicolon
(	left parenthesis	<	less than
)	right parenthesis	=	equals
*	asterisk	>	greater than
+	plus sign	?	question mark
,	comma	_	underline or underscore
|	vertical bar	ˆ	caret
!	exclamation mark		

MBCS Considerations

All multi-byte characters are treated as letters, except for the double-byte blank which is a special character.

Tokens

The basic syntactical units of the language are called *tokens*. A token is a sequence of one or more characters. A token cannot contain blank characters, unless it is a string constant or delimited identifier, which may contain blanks. (These terms are defined later.)

Tokens are classified as *ordinary* or *delimiter* tokens:

- An *ordinary token* is a numeric constant, an ordinary identifier, a host identifier, or a keyword.

  Examples

  ```
  1        .1        +2        SELECT        E        3
  ```

- A *delimiter token* is a string constant, a delimited identifier, an operator symbol, or any of the special characters shown in the syntax diagrams. A question mark is also a delimiter token when it serves as a parameter marker, as explained under "PREPARE" on page 954.

  Examples

  ```
     ,       'string'       "fld1"       =       .
  ```

Spaces: A space is a sequence of one or more blank characters. Tokens other than string constants and delimited identifiers must not include a space. Any token may be followed by a space. Every ordinary token must be followed by a space or a delimiter token if allowed by the syntax.

Comments: Static SQL statements may include host language comments or SQL comments. Either type of comment may be specified wherever a space may be specified, except within a delimiter token or between the keywords EXEC and SQL. SQL comments are introduced by two consecutive hyphens (--) and ended by the end of the line. For more information, see "SQL Comments" on page 463.

Uppercase and Lowercase: Any token may include lowercase letters, but a lowercase letter in an ordinary token is folded to uppercase, except for host variables in the C language, which has case-sensitive identifiers. Delimiter tokens are never folded to uppercase. Thus, the statement:

```
select * from EMPLOYEE where lastname = 'Smith';
```

is equivalent, after folding, to:

```
SELECT * FROM EMPLOYEE WHERE LASTNAME = 'Smith';
```

MBCS Considerations

Multi-byte alphabetic letters are not folded to uppercase. Single-byte characters, a to z, are folded to uppercase.

Identifiers

An *identifier* is a token that is used to form a name. An identifier in an SQL statement is either an SQL identifier or a host identifier.

SQL Identifiers

There are two types of SQL identifiers: *ordinary* and *delimited*

- An *ordinary identifier* is a letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier should not be identical to a reserved word (see "Appendix H. Reserved Schema Names and Reserved Words" on page 1279 for information on reserved words).

- A *delimited identifier* is a sequence of one or more characters enclosed within quotation marks ("). Two consecutive quotation marks are used to represent one quotation mark within the delimited identifier. In this way an identifier can include lowercase letters.

Examples of ordinary and delimited identifiers are:

```
   WKLYSAL    WKLY_SAL    "WKLY_SAL"    "WKLY SAL"    "UNION"    "wkly_sal"
```

Character conversions between identifiers created on a double-byte code page but used by an application or database on a multi-byte code page may require special consideration. After conversion to multi-byte, it is possible that such identifiers may exceed the length limit for an identifier (see “Appendix O. Japanese and Traditional-Chinese EUC Considerations” on page 1341 for details).

Host Identifiers

A *host identifier* is a name declared in the host program. The rules for forming a host identifier are the rules of the host language. A host identifier should not be greater than 255 characters and should not begin with upper or lower case spelling of ’SQL’ or ’DB2’.

Naming Conventions and Implicit Object Name Qualifications

The rules for forming a name depend on the type of the object designated by the name. Database object names may be made up of a single identifier or they may be schema qualified objects made up of two identifiers. Schema qualified object names may be specified without the schema name. In such cases, a schema name is implicit.

In dynamic SQL statements, a schema qualified object name implicitly uses the CURRENT SCHEMA special register value as the qualifier for unqualified object name references. By default it is set to the current authorization ID. See “SET SCHEMA” on page 1033 for details. If the dynamic SQL statement is from a package bound with the DYNAMICRULES BIND option, the CURRENT SCHEMA special register is not used in qualification. In a DYNAMICRULES BIND package, the package default qualifier is used as the value for qualification of unqualified object references within dynamic SQL statements.

In static SQL statements, the QUALIFIER precompile/bind option implicitly specifies the qualifier for unqualified database object names. By default, it is set to the package authorization ID. See the *Command Reference* for details.

The syntax diagrams use different terms for different types of names. The following list defines these terms. For maximum length of various identifiers refer to “Appendix A. SQL Limits” on page 1099.

alias-name	A schema qualified name that designates an alias.
attribute-name	An identifier that designates an attribute of a structured data type.
authorization-name	An identifier that designates a user or group. Note the following restrictions on the characters that can be used: • Valid characters are A through Z, a through z, 0 through 9, #, @, $ and _. • The name must not begin with the characters 'SYS', 'IBM' or 'SQL'. • The name must not be: ADMINS, GUESTS, LOCAL, PUBLIC, or USERS. • A delimited authorization ID must not contain lowercase letters.
bufferpool-name	An identifier that designates a bufferpool.
column-name	A qualified or unqualified name that designates a column of a table or view. The qualifier is a table-name, a view-name, a nickname, or a correlation-name.
condition-name	An identifier that designates a condition in an SQL procedure.
constraint-name	An identifier that designates a referential constraint, primary key constraint, unique constraint or a table check constraint.
correlation-name	An identifier that designates a result table.
cursor-name	An identifier that designates an SQL cursor. For host compatibility, a hyphen character may be used in the name.
data-source-name	A identifier that designates a data source. This identifier is the first of the three parts of remote-object-name.
descriptor-name	A colon followed by a host identifier that designates an SQL descriptor area (SQLDA). See “References to Host Variables” on page 135 for a description of a host identifier. Note that a descriptor-name never includes an indicator variable.
distinct-type-name	A qualified or unqualified name that

	designates a distinct type-name. An unqualified distinct-type-name in an SQL statement is implicitly qualified by the database manager, depending on context.
event-monitor-name	An identifier that designates an event monitor.
function-mapping-name	An identifier that designates a function mapping.
function-name	A qualified or unqualified name that designates a function. A unqualified function-name in an SQL statement is implicitly qualified by the database manager, depending on context.
group-name	An unqualified identifier that designates a transform group defined for a structured type.
host-variable	A sequence of tokens that designates a host variable. A host variable includes at least one host identifier, as explained in "References to Host Variables" on page 135.
index-name	A schema qualified name that designates an index or an index specification.
label	An identifier that designates a label in an SQL procedure.
method-name	An identifier that designates a method. The schema context for a method is determined by the schema of the subject type (or a supertype of the subject type) of the method.
nickname	A schema qualified name that designates a federated server reference to a table or view.
nodegroup-name	An identifier that designates a nodegroup.
package-name	A schema qualified name that designates a package.
parameter-name	An identifier that names a parameter that can be referenced in a procedure, user-defined function, method, or index extension.
procedure-name	A qualified or unqualified name that designates a procedure. An unqualified procedure-name in an SQL statement is implicitly qualified by the database manager, depending on context.

remote-authorization-name	An identifier that designates a data source user. The rules for authorization names vary from data source to data source.
remote-function-name	A name that designates a function registered to a data source database.
remote-object-name	A three-part name that designates a data source table or view and that identifies the data source where the table or view resides. The parts in this name are data-source-name, remote-schema-name, and remote-table-name.
remote-schema-name	A name that designates the schema that a data source table or view belongs. This name is the second of the three parts of remote-object-name.
remote-table-name	A name that designates a table or view at a data source. This name is the third of the three parts of remote-object-name.
remote-type-name	A data type supported by a data source database. Do not use the long form for system built-in types (for example, use CHAR instead of CHARACTER).
savepoint-name	An identifier that designates a savepoint.
schema-name	An identifier that provides a logical grouping for SQL objects. A schema-name used as a qualifier of the name of an object may be implicitly determined: • from the value of the CURRENT SCHEMA special register • from the value of the QUALIFIER precompile/bind option • based on a resolution algorithm that uses the CURRENT PATH special register • based on the schema name of another object in the same SQL statement. To avoid complications, it is recommended that the schema name ″SESSION″ not be used as a schema, except as the schema for declared global temporary tables (which must use the schema name ″SESSION″).
server-name	An identifier that designates an application

	server. In a federated system, server-name also designates the local name of a data source.
specific-name	A qualified or unqualified name that designates a specific name. An unqualified specific-name in an SQL statement is implicitly qualified by the database manager, depending on context.
SQL-variable-name	Defines the name of a local variable in an SQL procedure statement. SQL-variable-names can be used in other SQL statements where a host-variable name is allowed. The name can be qualified by the label of the compound statement that declared the SQL variable.
statement-name	An identifier that designates a prepared SQL statement.
supertype-name	A qualified or unqualified name that designates the supertype of a type-name. An unqualified supertype-name in an SQL statement is implicitly qualified by the database manager, depending on context.
table-name	A schema qualified name that designates a table.
tablespace-name	An identifier that designates a table space.
trigger-name	A schema qualified name that designates a trigger.
type-mapping-name	An identifier that designates a data type mapping.
type-name	A qualified or unqualified name that designates a type-name. An unqualified type-name in an SQL statement is implicitly qualified by the database manager, depending on context.
typed-table-name	A schema qualified name that designates a typed table.
typed-view-name	A schema qualified name that designates a typed view.
view-name	A schema qualified name that designates a view.
wrapper-name	An identifier that designates a wrapper.

Aliases

A table alias can be thought of as an alternative name for a table or view. A table or view, therefore, can be referred to in an SQL statement by its name or by a table alias.

An alias can be used wherever a table or view name can be used. An alias can be created even though the object does not exist (though it must exist by the time a statement referring to it is compiled). It can refer to another alias if no circular or repetitive references are made along the chain of aliases. An alias can only refer to a table, view, or alias within the same database. An alias name cannot be used where a new table or view name is expected, such as in the CREATE TABLE or CREATE VIEW statements; for example, if an alias name of PERSONNEL is created then a subsequent statement such as CREATE TABLE PERSONNEL... will cause an error.

The option of referring to a table or view by an alias is not explicitly shown in the syntax diagrams or mentioned in the description of the SQL statement.

A new unqualified alias cannot have the same fully-qualified name as an existing table, view, or alias.

The effect of using an alias in an SQL statement is similar to that of text substitution. The alias, which must be defined when the SQL statement is compiled, is replaced at statement compilation time by the qualified base table or view name. For example, if `PBIRD.SALES` is an alias for `DSPN014.DIST4_SALES_148`, then at compilation time:

```
SELECT * FROM PBIRD.SALES
```

effectively becomes

```
SELECT * FROM DSPN014.DIST4_SALES_148
```

In a federated system, the aforementioned uses and restrictions apply not only to table aliases but also to aliases for nicknames. Thus, a nickname's alias can be used in lieu of the nickname in an SQL statement; an alias can be created for a nickname that does not yet exist, provided that the nickname is created before statements that reference the alias are compiled; an alias for a nickname can refer to another alias for that nickname; and so on.

For syntax toleration of other relational database management system applications, SYNONYM can be used in place of ALIAS in the CREATE ALIAS and DROP ALIAS statements.

Refer to "CREATE ALIAS" on page 566 for more information about aliases.

Authorization IDs and authorization-names

An *authorization ID* is a character string that is obtained by the database manager when a connection is established between the database manager and either an application process or a program preparation process. It designates a set of privileges. It may also designate a user or a group of users, but this property is not controlled by the database manager.

Authorization IDs are used by the database manager to provide:

- Authorization checking of SQL statements
- Default value for the QUALIFIER precompile/bind option and the CURRENT SCHEMA special register. Also, the authorization ID is included in the default CURRENT PATH special register and FUNCPATH precompile/bind option.

An authorization ID applies to every SQL statement. The authorization ID that applies to a static SQL statement is the authorization ID that is used during program binding. The authorization ID that applies to a dynamic SQL statement is based on the DYNAMICRULES option supplied at bind time for the package issuing the dynamic SQL statement. For a package bound with DYNAMICRULES RUN, the authorization ID used is the authorization ID of the user executing the package. For a package bound with DYNAMICRULES BIND, the authorization ID used is the authorization ID of the package. This is called the *run-time authorization ID*.

An *authorization-name* specified in an SQL statement should not be confused with the authorization ID of the statement. An authorization-name is an identifier that is used within various SQL statement. An authorization-name is used in a CREATE SCHEMA statement to designate the owner of the schema. An authorization-name is used in GRANT and REVOKE statements to designate a target of the grant or revoke. Note that the premise of a grant of privileges to *X* is that *X* or a member of the group X will subsequently be the authorization ID of statements which require those privileges.

Examples:

- Assume SMITH is the userid and the authorization ID that the database manager obtained when the connection was established with the application process. The following statement is executed interactively:

  ```
  GRANT SELECT ON TDEPT TO KEENE
  ```

  SMITH is the authorization ID of the statement. Hence, in a dynamic SQL statement the default value of the CURRENT SCHEMA special register and in static SQL the default QUALIFIER precompile/bind option is SMITH. Thus, the authority to execute the statement is checked against SMITH and

SMITH is the *table-name* implicit qualifier based on qualification rules described in "Naming Conventions and Implicit Object Name Qualifications" on page 66.

KEENE is an authorization-name specified in the statement. KEENE is given the SELECT privilege on SMITH.TDEPT.

- Assume SMITH has administrative authority and is the authorization ID of the following dynamic SQL statements with no SET SCHEMA statement issued during the session:

  ```
  DROP TABLE TDEPT
  ```

  Removes the `SMITH.TDEPT` table.

  ```
  DROP TABLE SMITH.TDEPT
  ```

  Removes the `SMITH.TDEPT` table.

  ```
  DROP TABLE KEENE.TDEPT
  ```

  Removes the `KEENE.TDEPT` table. Note that `KEENE.TDEPT` and `SMITH.TDEPT` are different tables.

  ```
  CREATE SCHEMA PAYROLL AUTHORIZATION KEENE
  ```

  KEENE is the authorization-name specified in the statement which creates a schema called PAYROLL. KEENE is the owner of the schema PAYROLL and is given CREATEIN, ALTERIN, and DROPIN privileges with the ability to grant them to others.

Dynamic SQL Characteristics at run-time

The BIND and PRECOMPILE command option OWNER defines the authorization ID of the package.

The BIND and PRECOMPILE command option QUALIFIER defines implicit qualifier for unqualified objects contained in the package.

The BIND and PRECOMPILE command option DYNAMICRULES determines whether dynamic SQL statements are processed at run time with run time rules, DYNAMICRULES RUN, or with bind time rules, DYNAMICRULES BIND. These rules indicate the setting of the value used as authorization ID and for the setting of the value used for implicit qualification of unqualified object references within dynamic SQL statements. The DYNAMICRULES options have the effects described in the following tables:

Dynamic SQL Characteristics at run-time

Table 1. Static SQL Characteristics affected by OWNER and QUALIFIER

Feature	Only OWNER Specified	Only QUALIFIER Specified	QUALIFIER and OWNER specified
Authorization ID Used	ID of User specified in the OWNER option on the BIND command	ID of User Binding the Package	ID of User specified in the OWNER option on the BIND command
Unqualified Object Qualification Value Used	ID of User specified in the OWNER option on the BIND command	ID of User specified in the QUALIFIER option on the BIND command	ID of User specified in the QUALIFIER option on the BIND command

Table 2. Dynamic SQL Characteristics affected by DYNAMICRULES, OWNER and QUALIFIER

Feature	RUN	BIND
Authorization ID Used	ID of User Executing Package	Authorization ID for the package
Unqualified Object Qualification Value Used	CURRENT SCHEMA Special Register	Authorization ID for the package

Considerations regarding the DYNAMICRULES option:

- The CURRENT SCHEMA special register will not be used to qualify unqualified object references within dynamic SQL statements executed from a package bound with DYNAMICRULES BIND. Instead, DB2 will use the package default qualifier as is shown in the table. This is true even after you issue the statement SET CURRENT SCHEMA in order to change the CURRENT SCHEMA special register; the register value will be changed but not used.
- The following dynamically prepared SQL statements can not be used within a package bound with DYNAMICRULES BIND option: GRANT, REVOKE, ALTER, CREATE, DROP, COMMENT ON, RENAME, SET INTEGRITY, SET EVENT MONITOR STATE and queries that reference a nickname.
- If multiple packages are referenced during a single connection, dynamic SQL will behave according to the BIND options for the package in which a statement is bound.
- It is important to keep in mind that when binding a package with DYNAMICRULES BIND, the binder of the package should not have any authorities granted to them that you would not want the user of the package to have since a dynamic statement will be using the authorization ID of the package owner.

Authorization IDs and Statement Preparation

If VALIDATE BIND is specified at BIND time, the privileges required to manipulate tables and views must exist at bind time. If the privileges or the referenced objects do not exist and SQLERROR NOPACKAGE is in effect, the bind operation is unsuccessful. If SQLERROR CONTINUE is specified, then the bind is successful and any statements in error are flagged. Any attempt to execute a statement flagged as an error will result in an error in the application.

If a package is bound with VALIDATE RUN, all normal BIND processing is completed, but the privileges required to use the tables and views referenced in the application need not exist at that time. If any privilege required for a statement does not exist at bind time, an incremental bind is performed whenever the statement is first executed within an application, and all privileges required for the statement must exist. If any privilege does not exist, execution of the statement is unsuccessful. When the authorization check is performed at run time, it is performed using the package owner's authorization ID.

Data Types

For information about specifying the data types of columns, see "CREATE TABLE" on page 712.

The smallest unit of data that can be manipulated in SQL is called a *value.* How values are interpreted depends on the data type of their source. The sources of values are:

- Constants
- Columns
- Host variables
- Functions
- Expressions
- Special registers.

DB2 supports a number of built-in datatypes, which are described in this section. It also provides support for user-defined data types. See "User Defined Types" on page 87 for a description of user-defined data types.

Figure 10 on page 76 illustrates the supported built-in data types.

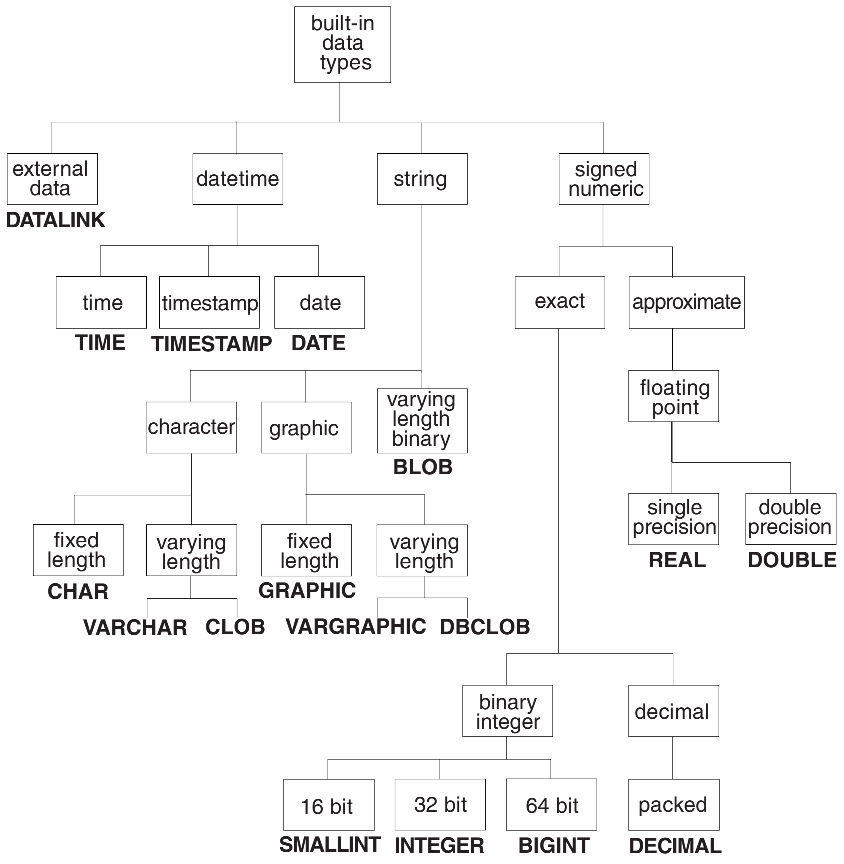


Figure 10. Supported Built-in Data Types

Nulls

All data types include the null value. The null value is a special value that is distinct from all non-null values and thereby denotes the absence of a (non-null) value. Although all data types include the null value, columns defined as NOT NULL cannot contain null values.

Large Objects (LOBs)

The term *large object* and the generic acronym *LOB* are used to refer to any BLOB, CLOB, or DBCLOB data type. LOB values are subject to the restrictions that apply to LONG VARCHAR values as specified in "Restrictions Using Varying-Length Character Strings" on page 79. For LOB strings, these restrictions apply even when the length attribute of the string is 254 bytes or less.

Character Large Object (CLOB) Strings
A *Character Large OBject (CLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. A *CLOB* is used to store large SBCS or mixed (SBCS and MBCS) character-based data such as documents written with a single character set (and, therefore, has an SBCS or mixed code page associated with it). Note that a CLOB is considered to be a character string.

Double-Byte Character Large Object (DBCLOB) Strings
A *Double-Byte Character Large OBject (DBCLOB)* is a varying-length string of double-byte characters that can be up to 1 073 741 823 characters long. A *DBCLOB* is used to store large DBCS character based data such as documents written with a single character set (and, therefore has a DBCS CCSID associated with it). Note that a DBCLOB is considered to be a graphic string.

Binary Large Objects (BLOBs)
A *Binary Large OBject (BLOB)* is a varying-length string measured in bytes that can be up to 2 gigabytes (2 147 483 647 bytes) long. A *BLOB* is primarily intended to hold non-traditional data such as pictures, voice, and mixed media. Another use is to hold structured data for exploitation by user-defined types and user-defined functions. As with FOR BIT DATA character strings, BLOB strings are not associated with a character set.

Manipulating Large Objects (LOBs) with Locators
Since LOB values can be very large, the transfer of these values from the database server to client application program host variables can be time consuming. However, it is also true that application programs typically process LOB values a piece at a time, rather than as a whole. For those cases where an application does not need (or want) the entire LOB value to be stored in application memory, the application can reference a LOB value via a large object locator (LOB locator).

A *large object locator* or LOB locator is a host variable with a value that represents a single LOB value in the database server. LOB locators were developed to provide users with a mechanism by which they could easily manipulate very large objects in application programs without requiring them to store the entire LOB value on the client machine where the application program may be running.

For example, when selecting a LOB value, an application program could select the entire LOB value and place it into an equally large host variable (which is acceptable if the application program is going to process the entire LOB value at once), or it could instead select the LOB value into a LOB locator. Then, using the LOB locator, the application program can issue subsequent database operations on the LOB value (such as applying the scalar functions SUBSTR, CONCAT, VALUE, LENGTH, doing an assignment, searching the LOB with LIKE or POSSTR, or applying UDFs against the LOB) by supplying the locator value as input. The resulting output of the locator operation, for example the amount of data assigned to a client host variable, would then typically be a small subset of the input LOB value.

LOB locators may also represent more than just base values; they can also represent the value associated with a LOB expression. For example, a LOB locator might represent the value associated with:

```
SUBSTR( <lob 1> CONCAT <lob 2> CONCAT <lob 3>, <start>, <length> )
```

For normal host variables in an application program, when a null value is selected into that host variable, the indicator variable is set to -1, signifying that the value is null. In the case of LOB locators, however, the meaning of indicator variables is slightly different. Since a locator host variable itself can never be null, a negative indicator variable value indicates that the LOB value represented by the LOB locator is null. The null information is kept local to the client by virtue of the indicator variable value — the server does not track null values with valid locators.

It is important to understand that a LOB locator represents a value, not a row or location in the database. Once a value is selected into a locator, there is no operation that one can perform on the original row or table that will affect the value which is referenced by the locator. The value associated with a locator is valid until the transaction ends, or until the locator is explicitly freed, whichever comes first. Locators do not force extra copies of the data in order to provide this function. Instead, the locator mechanism stores a description of the base LOB value. The materialization of the LOB value (or expression, as shown above) is deferred until it is actually assigned to some location — either into a user buffer in the form of a host variable or into another record's field value in the database.

A LOB locator is only a mechanism used to refer to a LOB value during a transaction; it does not persist beyond the transaction in which it was created. Also, it is not a database type; it is never stored in the database and, as a result, cannot participate in views or check constraints. However, since a locator is a client representation of a LOB type, there are SQLTYPEs for LOB locators so that they can be described within an SQLDA structure that is used by FETCH, OPEN and EXECUTE statements.

Character Strings

A *character string* is a sequence of bytes. The length of the string is the number of bytes in the sequence. If the length is zero, the value is called the *empty string*. This value should not be confused with the null value.

Fixed-Length Character Strings
All values of a fixed-length string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 254, inclusive.

Varying-Length Character Strings
Varying-length character strings are of three types: VARCHAR, LONG VARCHAR, and CLOB.

- VARCHAR types are varying-length strings of up to 32 672 bytes.
- LONG VARCHAR types are varying-length strings of up to 32 700 bytes.
- CLOB types are varying-length strings of up to 2 gigabytes.

Restrictions Using Varying-Length Character Strings: Special restrictions apply to an expression resulting in a varying-length string data type whose maximum length is greater than 255 bytes; such expressions are not permitted in:

- A SELECT list preceded by DISTINCT
- A GROUP BY clause
- An ORDER BY clause
- A column function with DISTINCT
- A subselect of a set operator other than UNION ALL.

In addition to the restrictions listed above, expressions resulting in LONG VARCHAR, CLOB data types or structured type columns are not permitted in:

- A Basic, Quantified, BETWEEN, or IN predicate
- A column function
- VARGRAPHIC, TRANSLATE, and datetime scalar functions
- The pattern operand in a LIKE predicate or the search string operand in a POSSTR function
- The string representation of a datetime value

The functions in the SYSFUN schema taking a VARCHAR as an argument will not accept VARCHARs greater than 4 000 bytes long as an argument. However, many of these functions also have an alternative signature accepting a CLOB(1M). For these functions the user may explicitly cast the greater than 4 000 VARCHAR strings into CLOBs and then recast the result back into VARCHARs of desired length.

NUL-Terminated Character Strings
NUL-terminated character strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Development Guide* for more information on the treatment of NUL-terminated character strings.

Character Subtypes

Each character string is further defined as one of:

Bit data	Data that is not associated with a code page.
SBCS data	Data in which every character is represented by a single byte.
Mixed data	Data that may contain a mixture of characters from a single-byte character set (SBCS) and a multi-byte character set (MBCS).

SBCS and MBCS Considerations: SBCS data is supported only in a SBCS database. Mixed data is only supported in an MBCS database.

Graphic Strings

A *graphic string* is a sequence of bytes which represents double-byte character data. The length of the string is the number of double-byte characters in the sequence. If the length is zero, the value is called the empty string. This value should not be confused with the null value.

Graphic strings are not validated to ensure that their values contain only double-byte character code points. [15] Rather, the database manager assumes that double-byte character data is contained within graphic data fields. The database manager checks that a graphic string value is an even number of bytes in length.

A graphic string data type may be fixed length or varying length; the semantics of fixed length and varying length are analogous to those defined for character string data types.

Fixed-Length Graphic Strings

All values of a fixed-length graphic string column have the same length, which is determined by the length attribute of the column. The length attribute must be between 1 and 127, inclusive.

Varying-Length Graphic Strings

Varying-length graphic strings are of three types: VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB.

- VARGRAPHIC types are varying-length strings of up to 16 336 double-byte characters.
- LONG VARGRAPHIC types are varying-length strings of up to 16 350 double-byte characters.
- DBCLOB types are varying-length strings of up to 1 073 741 823 double-byte characters.

15. The exception to this rule is an application precompiled with the WCHARTYPE CONVERT option. In this case, validation does occur. See “Programming in C and C++” in the *Application Development Guide* for details.

Special restrictions apply to an expression resulting in a varying-length graphic string data type whose maximum length is greater than 127. Those restrictions are the same as specified in “Restrictions Using Varying-Length Character Strings” on page 79.

NUL-Terminated Graphic Strings

NUL-terminated graphic strings found in C are handled differently, depending on the standards level of the precompile option. See the C language specific section in the *Application Development Guide* for more information on the treatment of NUL-terminated graphic strings.

This data type cannot be created in a table. It can only be used to insert data into and retrieve data from the database.

Binary String

A *binary string* is a sequence of bytes. Unlike a character string which usually contains text data, a binary string is used to hold non-traditional data such as pictures. Note that character strings of the ’bit data’ subtype may be used for similar purposes, but the two data types are not compatible. The BLOB scalar function can be used to cast a character for bit string to a binary string. The length of a binary string is the number of bytes. It is not associated with a code page. Binary strings have the same restrictions as character strings (see “Restrictions Using Varying-Length Character Strings” on page 79 for details).

Numbers

All numbers have a sign and a precision. The *precision* is the number of bits or digits excluding the sign. The sign is considered positive if the value of a number is zero.

Small Integer (SMALLINT)

A *small integer* is a two byte integer with a precision of 5 digits. The range of small integers is -32 768 to 32 767.

Large Integer (INTEGER)

A *large integer* is a four byte integer with a precision of 10 digits. The range of large integers is −2 147 483 648 to +2 147 483 647.

Big Integer (BIGINT)

A *big integer* is an eight byte integer with a precision of 19 digits. The range of big integers is −9 223 372 036 854 775 808 to +9 223 372 036 854 775 807.

Single-Precision Floating-Point (REAL)

A *single-precision floating-point* number is a 32 bit approximation of a real number. The number can be zero or can range from -3.402E+38 to -1.175E-37, or from 1.175E-37 to 3.402E+38.

Double-Precision Floating-Point (DOUBLE or FLOAT)

A *double-precision floating-point* number is a 64 bit approximation of a real number. The number can be zero or can range from -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

Decimal (DECIMAL or NUMERIC)

A *decimal* value is a packed decimal number with an implicit decimal point. The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits. For information on packed decimal representation, see "Packed Decimal Numbers" on page 1124.

All values of a decimal column have the same precision and scale. The range of a decimal variable or the numbers in a decimal column is $-n$ to $+n$, where the absolute value of n is the largest number that can be represented with the applicable precision and scale. The maximum range is -10**31+1 to 10**31-1.

Datetime Values

The datetime data types are described below. Although datetime values can be used in certain arithmetic and string operations and are compatible with certain strings, they are neither strings nor numbers.

Date

A *date* is a three-part value (year, month, and day). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to x, where x depends on the month.

The internal representation of a date is a string of 4 bytes. Each byte consists of 2 packed decimal digits. The first 2 bytes represent the year, the third byte the month, and the last byte the day.

The length of a DATE column, as described in the SQLDA, is 10 bytes, which is the appropriate length for a character string representation of the value.

Time

A *time* is a three-part value (hour, minute, and second) designating a time of day under a 24-hour clock. The range of the hour part is 0 to 24; while the range of the other parts is 0 to 59. If the hour is 24, the minute and second specifications will be zero.

The internal representation of a time is a string of 3 bytes. Each byte is 2 packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

The length of a TIME column, as described in the SQLDA, is 8 bytes, which is the appropriate length for a character string representation of the value.

Timestamp
A *timestamp* is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined above, except that the time includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes, each of which consists of 2 packed decimal digits. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds.

The length of a TIMESTAMP column, as described in the SQLDA, is 26 bytes, which is the appropriate length for the character string representation of the value.

String Representations of Datetime Values
Values whose data types are DATE, TIME, or TIMESTAMP are represented in an internal form that is transparent to the SQL user. Dates, times, and timestamps can, however, also be represented by character strings, and these representations directly concern the SQL user since there are no constants or variables whose data types are DATE, TIME, or TIMESTAMP. Thus, to be retrieved, a datetime value must be assigned to a character string variable. Note that the CHAR function can be used to change a datetime value to a string representation. The character string representation is normally the default format of datetime values associated with the country code of the database, unless overridden by specification of the *DATETIME* option when the program is precompiled or bound to the database.

No matter what its length, a large object string or LONG VARCHAR cannot be used as the string that represents a datetime value; otherwise an error is raised (SQLSTATE 42884).

When a valid string representation of a datetime value is used in an operation with an internal datetime value, the string representation is converted to the internal form of the date, time, or timestamp before the operation is performed. The following sections define the valid string representations of datetime values.

Date Strings
A string representation of a date is a character string that starts with a digit and has a length of at least 8 characters. Trailing blanks may be included; leading zeros may be omitted from the month and day portions.

Valid string formats for dates are listed in Table 1. Each format is identified by name and includes an associated abbreviation and an example of its use.

Table 3. Formats for String Representations of Dates

Format Name	Abbreviation	Date Format	Example
International Standards Organization	ISO	yyyy-mm-dd	1991-10-27
IBM USA standard	USA	mm/dd/yyyy	10/27/1991
IBM European standard	EUR	dd.mm.yyyy	27.10.1991
Japanese Industrial Standard Christian era	JIS	yyyy-mm-dd	1991-10-27
Site-defined (see *DB2 Data Links Manager Quick Beginnings*)	LOC	Depends on database country code	—

Time Strings

A string representation of a time is a character string that starts with a digit and has a length of at least 4 characters. Trailing blanks may be included; a leading zero may be omitted from the hour part of the time and seconds may be omitted entirely. If seconds are omitted, an implicit specification of 0 seconds is assumed. Thus, 13.30 is equivalent to 13.30.00.

Valid string formats for times are listed in Table 4. Each format is identified by name and includes an associated abbreviation and an example of its use.

Table 4. Formats for String Representations of Times

Format Name	Abbreviation	Time Format	Example
International Standards Organization[2]	ISO	hh.mm.ss	13.30.05
IBM USA standard	USA	hh:mm AM or PM	1:30 PM
IBM European standard	EUR	hh.mm.ss	13.30.05
Japanese Industrial Standard Christian Era	JIS	hh:mm:ss	13:30:05
Site-defined (see *DB2 Data Links Manager Quick Beginnings*)	LOC	Depends on database country code	—

Notes:

1. In ISO, EUR and JIS format, .ss (or :ss) is optional.
2. The International Standards Organization recently changed the time format so that it is identical with the Japanese Industrial Standard Christian Era. Therefore, use JIS format if an application requires the current International Standards Organization format.

3. In the case of the USA time string format, the minutes specification may be omitted, indicating an implicit specification of 00 minutes. Thus 1 PM is equivalent to 1:00 PM.
4. In the USA time format, the hour must not be greater than 12 and cannot be 0 except for the special case of 00:00 AM. There is a single space before the AM and PM. Using the ISO format of the 24-hour clock, the correspondence between the USA format and the 24-hour clock is as follows:
   12:01 AM through 12:59 AM corresponds to 00.01.00 through 00.59.00.
   01:00 AM through 11:59 AM corresponds to 01.00.00 through 11.59.00.
   12:00 PM (noon) through 11:59 PM corresponds to 12.00.00 through 23.59.00.
   12:00 AM (midnight) corresponds to 24.00.00 and 00:00 AM (midnight) corresponds to 00.00.00.

Timestamp Strings

A string representation of a timestamp is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnn*. Trailing blanks may be included. Leading zeros may be omitted from the month, day, and hour part of the timestamp, and microseconds may be truncated or entirely omitted. If any trailing zero digits are omitted in the microseconds portion, an implicit specification of 0 is assumed for the missing digits. Thus, 1991-3-2-8.30.00 is equivalent to 1991-03-02-08.30.00.000000.

SQL statements also support the ODBC string representation of a timestamp as an input value only. The ODBC string representation of a timestamp has the form *yyyy-mm-dd hh:mm:ss.nnnnnn*. See the *CLI Guide and Reference* for more information on ODBC.

MBCS Considerations

Date, time and timestamp strings must contain only single-byte characters and digits.

DATALINK Values

A DATALINK value is an encapsulated value that contains a logical reference from the database to a file stored outside the database. The attributes of this encapsulated value are as follows:

link type
The currently supported type of link is 'URL' (Uniform Resource Locator).

data location
The location of a file linked with a reference within DB2, in the form of a URL. The allowed scheme names for this URL are:

- HTTP
- FILE

- UNC
- DFS

The other parts of the URL are:

- the file server name for the HTTP, FILE, and UNC schemes
- the cell name for the DFS scheme
- the full file path name within the file server or cell

See “Appendix P. BNF Specifications for DATALINKs” on page 1349 for more information on exact BNF (Backus Naur form) specifications for DATALINKs.

comment
Up to 254 bytes of descriptive information. This is intended for application specific uses such as further or alternative identification of the location of the data.

Leading and trailing blank characters are trimmed while parsing data location attributes as URLs. Also, the scheme names ('http', 'file', 'unc', 'dfs') and host are case-insensitive and are always stored in the database in uppercase. When a DATALINK value is fetched from a database, an access token is embedded within the URL attribute when appropriate. It is generated dynamically and is not a permanent part of the DATALINK value stored in the database. For more details, see the DATALINK related scalar functions, beginning with “DLCOMMENT” on page 287.

It is possible for a DATALINK value to have only a comment attribute and an empty data location attribute. Such a value may even be stored in a column but, of course, no file will be linked to such a column. The total length of the comment and the data location attribute of a DATALINK value is currently limited to 200 bytes.

It should be noted that DATALINKs cannot be exchanged with a DRDA server.

It is important to distinguish between these DATALINK references to files and the LOB file reference variables described in the section entitled “References to BLOB, CLOB, and DBCLOB Host Variables” on page 137. The similarity is that they both contain a representation of a file. However:

- DATALINKs are retained in the database and both the links and the data in the linked files can be considered as a natural extension of data in the database.
- File reference variables exist temporarily on the client and they can be considered as an alternative to a host program buffer.

Built-in scalar functions are provided to build a DATALINK value (DLVALUE) and to extract the encapsulated values from a DATALINK value (DLCOMMENT, DLLINKTYPE, DLURLCOMPLETE, DLURLPATH, DLURLPATHONLY, DLURLSCHEME, DLURLSERVER).

User Defined Types

Distinct Types

A *distinct type* is a user-defined data type that shares its internal representation with an existing type (its "source" type), but is considered to be a separate and incompatible type for most operations. For example, one might want to define a picture type, a text type, and an audio type, all of which have quite different semantics, but which use the built-in data type BLOB for their internal representation.

The following example illustrates the creation of a distinct type named AUDIO:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M)
```

Although AUDIO has the same representation as the built-in data type BLOB, it is considered to be a separate type that is not comparable to a BLOB or to any other type. This allows the creation of functions written specifically for AUDIO and assures that these functions will not be applied to any other type (pictures, text, etc.).

Distinct types are identified by qualified identifiers. If the schema name is not used to qualify the distinct type name when used in other than the CREATE DISTINCT TYPE, DROP DISTINCT TYPE, or COMMENT ON DISTINCT TYPE statements, the *SQL path* is searched in sequence for the first schema with a distinct type that matches. The SQL path is described in "CURRENT PATH" on page 122.

Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

Distinct types sourced on LONG VARCHAR, LONG VARGRAPHIC, LOB types, or DATALINK are subject to the same restrictions as their source type.

However, certain functions and operators of the source type can be explicitly specified to apply to the distinct type by defining user-defined functions that are sourced on functions defined on the source type of the distinct type (see "User-defined Type Comparisons" on page 106 for examples). The comparison operators are automatically generated for user-defined distinct types, except those using LONG VARCHAR, LONG VARGRAPHIC, BLOB, CLOB, DBCLOB, or DATALINK as the source type. In addition, functions are generated to support casting from the source type to the distinct type and from the distinct type to the source type.

Structured Types

A *structured type* is a user-defined data type that has a structure that is defined in the database. It contains a sequence of named *attributes*, each of which has a data type. A structured type also includes a set of method specifications.

A structured type may be used as the type of a table, view, or column. When used as a type for a table or view, that table or view is known as a *typed table* or *typed view*, respectively. For typed tables and typed views, the names and data types of the attributes of the structured type become the names and data types of the columns of this typed table or typed view. Rows of the typed table or typed view can be thought of as a representation of instances of the structured type. When used as a data type for a column, the column contains values of that structured type (or values of any of that type's subtypes, as defined below). Methods are used to retrieve or manipulate attributes of a structured column object.

Terminology: A *supertype* is a structured type for which other structured types, called *subtypes*, have been defined. A subtype inherits all the attributes and methods of its supertype and may have additional attributes and methods defined. The set of structured types that are related to a common supertype is called a *type hierarchy* and the type that does not have any supertype is called the *root type* of the type hierarchy.

The term subtype applies to a user-defined structured type and all user-defined structured types that are below it in the type hierarchy. Therefore, a subtype of a structured type T is T and all structured types below T in the hierarchy. A *proper subtype* of a structured type T is a structured type below T in the type hierarchy.

There are restrictions on having recursive type definitions in a type hierarchy. For this reason, it is necessary to develop a shorthand way of referring to the specific type of recursive definitions that are allowed. The following definitions are used:

- *Directly uses*: A type **A** is said to directly use another type **B**, if and only if one of the following is true:
  1. type **A** has an attribute of type **B**
  2. type **B** is a subtype of **A**, or a supertype of **A**
- *Indirectly uses*: A type **A** is said to indirectly use a type **B**, if one of the following is true:

1. type **A** directly uses type **B**
2. type **A** directly uses some type **C**, and type **C** indirectly uses type **B**

A type may not be defined so that one of its attribute types directly or indirectly uses itself. If it is necessary to have such a configuration, consider using a reference as the attribute. For example, with structured type attributes, there cannot be an instance of ″employee″ with an attribute of ″manager″ when ″manager″ is of type ″employee″. There can, however, be an attribute of ″manager″ with a type of REF(employee).

A type cannot be dropped when certain other objects use the type, either directly or indirectly. For example, a type cannot be dropped if a table or view column makes a direct or indirect use of the type. See Table 27 on page 885 for all objects that could restrict the dropping of types.

Structured type column values are subject to the restrictions that apply to CLOB values as specified in “Restrictions Using Varying-Length Character Strings” on page 79.

Reference (REF) Types

A *reference type* is a companion type to a structured type. Similar to a distinct type, a reference type is a scalar type that shares a common representation with one of the built-in data types. This same representation is shared for all types in the type hierarchy. The reference type representation is defined when the root type of a type hierarchy is created. When using a reference type, a structured type is specified as a parameter of the type. This parameter is called the *target type* of the reference.

The target of a reference is always a row in a typed table or view. When a reference type is used, it may have a *scope* defined. The scope identifies a table (called the *target table*) or view (called the *target view*) that contains the target row of a reference value. The target table or view must have the same type as the target type of the reference type. An instance of a scoped reference type uniquely identifies a row in a typed table or typed view, called the *target row*.

Promotion of Data Types

Data types can be classified into groups of related data types. Within such groups, a precedence order exists where one data type is considered to precede another data type. This precedence is used to allow the *promotion* of one data type to a data type later in the precedence ordering. For example, the data type CHAR can be promoted to VARCHAR; INTEGER can be promoted to DOUBLE PRECISION; but CLOB is NOT promotable to VARCHAR.

Promotion of data types is used when:
- performing function resolution (see "Function Resolution" on page 144)
- casting user-defined types (see "Casting Between Data Types" on page 91)
- assigning user-defined types to built-in data types (see "User-defined Type Assignments" on page 101).

Table 5 shows the precedence list (in order) for each data type and can be used to determine the data types to which a given data type can be promoted. The table shows that the best choice is always the same data type instead of choosing to promote to another data type.

Table 5. Data Type Precedence Table

Data Type	**Data Type Precedence List (in best-to-worst order)**
CHAR	CHAR, VARCHAR, LONG VARCHAR, CLOB
VARCHAR	VARCHAR, LONG VARCHAR, CLOB
LONG VARCHAR	LONG VARCHAR, CLOB
GRAPHIC	GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, DBCLOB
VARGRAPHIC	VARGRAPHIC, LONG VARGRAPHIC, DBCLOB
LONG VARGRAPHIC	LONG VARGRAPHIC, DBCLOB
BLOB	BLOB
CLOB	CLOB
DBCLOB	DBCLOB
SMALLINT	SMALLINT, INTEGER, BIGINT, decimal, real, double
INTEGER	INTEGER, BIGINT, decimal, real, double
BIGINT	BIGINT, decimal, real, double
decimal	decimal, real, double
real	real, double
double	double

Table 5. Data Type Precedence Table (continued)

Data Type	**Data Type Precedence List (in best-to-worst order)**
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
DATALINK	DATALINK
udt	udt (same name) or a supertype of udt
REF(T)	REF(S) (provided that S is a supertype of T)

Note:

The lower case types above are defined as follows:

decimal	= DECIMAL(p,s) or NUMERIC(p,s)
real	= REAL or FLOAT(*n*) where *n* is not greater than 24
double	= DOUBLE, DOUBLE PRECISION, FLOAT or FLOAT(*n*) where *n* is greater than 24
udt	= a user-defined type

Shorter and longer form synonyms of the data types listed are considered to be the same as the synonym listed.

Casting Between Data Types

There are many occasions where a value with a given data type needs to be *cast* to a different data type or to the same data type with a different length, precision or scale. Data type promotion (as defined in "Promotion of Data Types" on page 90) is one example where the promotion of one data type to another data type requires that the value is cast to the new data type. A data type that can be cast to another data type is *castable* from the source data type to the target data type.

Casting between data types can be done explicitly using the CAST specification (see "CAST Specifications" on page 173) but may also occur implicitly during assignments involving a user-defined types (see "User-defined Type Assignments" on page 101). Also, when creating sourced user-defined functions (see "CREATE FUNCTION" on page 589), the data types of the parameters of the source function must be castable to the data types of the function that is being created.

The supported casts between built-in data types are shown in Table 6 on page 93.

The following casts involving distinct types are supported:

- cast from distinct type *DT* to its source data type *S*
- cast from the source data type *S* of distinct type *DT* to distinct type *DT*
- cast from distinct type *DT* to the same distinct type *DT*
- cast from a data type *A* to distinct type *DT* where *A* is promotable to the source data type *S* of distinct type *DT* (see "Promotion of Data Types" on page 90)
- cast from an INTEGER to distinct type *DT* with a source data type SMALLINT
- cast from a DOUBLE to distinct type *DT* with a source data type REAL
- cast from a VARCHAR to distinct type *DT* with a source data type CHAR
- cast from a VARGRAPHIC to distinct type *DT* with a source data type GRAPHIC.

It is not possible to cast a structured type value to something else. A structured type *ST* should not need to be cast to one of its supertypes, since all methods on the supertypes of *ST* are applicable to *ST*. If the desired operation is only applicable to a subtype of *ST*, then use the subtype-treatment expression to treat *ST* as one of its subtypes. See "Subtype Treatment" on page 184 for details.

When a user-defined data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name. The SQL path is described further in "CURRENT PATH" on page 122.

The following casts involving reference types are supported:

- cast from reference type *RT* to its representation data type *S*
- cast from the representation data type *S* of reference type *RT* to reference type *RT*
- cast from reference type *RT* with target type *T* to a reference type *RS* with target type *S* where *S* is a supertype of *T*.
- cast from a data type *A* to reference type *RT* where *A* is promotable to the representation data type *S* of reference type *RT* (see "Promotion of Data Types" on page 90).

When the target type of a reference data type involved in a cast is not qualified by a schema name, the *SQL path* is used to find the first schema that includes the user-defined data type by that name. The SQL path is described further in "CURRENT PATH" on page 122.

Table 6. Supported Casts between Built-in Data Types

Target Data Type → Source Data Type ↓	SMALLINT	INTEGER	BIGINT	DECIMAL	REAL	DOUBLE	CHAR	VARCHAR	LONGVARCHAR	CLOB	GRAPHIC	VARGRAPHIC	LONGVARG	DBCLOB	DATE	TIME	TIMESTAMP	BLOB
SMALLINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
INTEGER	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
BIGINT	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
DECIMAL	Y	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-
REAL	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
DOUBLE	Y	Y	Y	Y	Y	Y	-	-	-	-	-	-	-	-	-	-	-	-
CHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	-	Y	-	-	Y	Y	Y	Y
VARCHAR	Y	Y	Y	Y	-	-	Y	Y	Y	Y	-	Y	-	-	Y	Y	Y	Y
LONG VARCHAR	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-	-	-	-	Y
CLOB	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	-	-	-	-	Y
GRAPHIC	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y
VARGRAPHIC	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y
LONG VARG	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y
DBCLOB	-	-	-	-	-	-	-	-	-	-	Y	Y	Y	Y	-	-	-	Y
DATE	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	-	-	-
TIME	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	-	Y	-	-
TIMESTAMP	-	-	-	-	-	-	Y	Y	-	-	-	-	-	-	Y	Y	Y	-
BLOB	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y

Notes

- See the description preceding the table for information on supported casts involving user-defined types and reference types.
- Only a DATALINK type can be cast to a DATALINK type.
- It is not possible to cast a structured type value to anything else.

Assignments and Comparisons

The basic operations of SQL are assignment and comparison. Assignment operations are performed during the execution of INSERT, UPDATE, FETCH, SELECT INTO, VALUES INTO and SET transition-variable statements. Arguments of functions are also assigned when invoking a function. Comparison operations are performed during the execution of statements that include predicates and other language elements such as MAX, MIN, DISTINCT, GROUP BY, and ORDER BY.

The basic rule for both operations is that the data type of the operands involved must be compatible. The compatibility rule also applies to set operations (see "Rules for Result Data Types" on page 107). The compatibility matrix is as follows.

Table 7. Data Type Compatibility for Assignments and Comparisons

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	Binary String	UDT
Binary Integer	Yes	Yes	Yes	No	No	No	No	No	No	[2]
Decimal Number	Yes	Yes	Yes	No	No	No	No	No	No	[2]
Floating Point	Yes	Yes	Yes	No	No	No	No	No	No	[2]
Character String	No	No	No	Yes	No	[1]	[1]	[1]	No [3]	[2]
Graphic String	No	No	No	No	Yes	No	No	No	No	[2]
Date	No	No	No	[1]	No	Yes	No	No	No	[2]
Time	No	No	No	[1]	No	No	Yes	No	No	[2]
Timestamp	No	No	No	[1]	No	No	No	Yes	No	[2]
Binary String	No	No	No	No [3]	No	No	No	No	Yes	[2]
UDT	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	[2]	Yes

Table 7. Data Type Compatibility for Assignments and Comparisons (continued)

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	Binary String	UDT

Note:

1 The compatibility of datetime values and character strings is limited to assignment and comparison:
- Datetime values can be assigned to character string columns and to character string variables as explained in “Datetime Assignments” on page 99.
- A valid string representation of a date can be assigned to a date column or compared with a date.
- A valid string representation of a time can be assigned to a time column or compared with a time.
- A valid string representation of a timestamp can be assigned to a timestamp column or compared with a timestamp.

2 A user-defined distinct type (UDDT) value is only comparable to a value defined with the same UDDT. In general, assignments are supported between a distinct type value and its source data type. A user-defined structured type is not comparable and can only be assigned to an operand of the same structured type or one of its supertypes. For additional information see “User-defined Type Assignments” on page 101.

3 Note that this means that character strings defined with the FOR BIT DATA attribute are also not compatible with binary strings.

4 A DATALINK operand can only be assigned to another DATALINK operand. The DATALINK value can only be assigned to a column if the column is defined with NO LINK CONTROL or the file exists and is not already under file link control.

5 For information on assignment and comparison of reference types see “Reference Type Assignments” on page 102 and “Reference Type Comparisons” on page 107.

A basic rule for assignment operations is that a null value cannot be assigned to a column that cannot contain null values, nor to a host variable that does not have an associated indicator variable. (See “References to Host Variables” on page 135 for a discussion of indicator variables.)

Numeric Assignments

The basic rule for numeric assignments is that the whole part of a decimal or integer number is never truncated. If the scale of the target number is less than the scale of the assigned number the excess digits in the fractional part of a decimal number are truncated.

Decimal or Integer to Floating-Point

Floating-point numbers are approximations of real numbers. Hence, when a decimal or integer number is assigned to a floating-point column or variable, the result may not be identical to the original number.

Floating-Point or Decimal to Integer

When a floating-point or decimal number is assigned to an integer column or variable, the fractional part of the number is lost.

Decimal to Decimal

When a decimal number is assigned to a decimal column or variable, the number is converted, if necessary, to the precision and the scale of the target. The necessary number of leading zeros is appended or eliminated, and, in the fractional part of the number, the necessary number of trailing zeros is appended, or the necessary number of trailing digits is eliminated.

Integer to Decimal

When an integer is assigned to a decimal column or variable, the number is converted first to a temporary decimal number and then, if necessary, to the precision and scale of the target. The precision and scale of the temporary decimal number is 5,0 for a small integer, or 11,0 for a large integer, or 19,0 for a big integer.

Floating-Point to Decimal

When a floating-point number is converted to decimal, the number is first converted to a temporary decimal number of precision 31, and then, if necessary, truncated to the precision and scale of the target. In this conversion, the number is rounded (using floating-point arithmetic) to a precision of 31 decimal digits. As a result, a number less than $0.5*10^{-31}$ is reduced to 0. The scale is given the largest possible value that allows the whole part of the number to be represented without loss of significance.

String Assignments

There are two types of assignments:

- *storage assignment* is when a value is assigned to a column or parameter of a function
- *retrieval assignment* is when a value is assigned to a host variable.

The rules for string assignment differ based on the assignment type.

Storage Assignment

The basic rule is that the length of the string assigned to a column or function parameter must not be greater than the length attribute of the column or the function parameter. When the length of the string is greater than the length attribute of the column or the function parameter, the following actions may occur:

- the string is assigned with trailing blanks truncated (from all string types except long strings) to fit the length attribute of the target column or function parameter
- an error is returned (SQLSTATE 22001) when:
  - non-blank characters would be truncated from other than a long string

– any character (or byte) would be truncated from a long string.

When a string is assigned to a fixed-length column and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of blanks. The pad character is always a blank even for columns defined with the FOR BIT DATA attribute.

Retrieval Assignment

The length of a string assigned to a host variable may be longer than the length attribute of the host variable. When a string is assigned to a host variable and the length of the string is longer than the length attribute of the variable, the string is truncated on the right by the necessary number of characters (or bytes). When this occurs, a warning is returned (SQLSTATE 01004) and the value 'W' is assigned to the SQLWARN1 field of the SQLCA.

Furthermore, if an indicator variable is provided, and the source of the value is not a LOB, the indicator variable is set to the original length of the string.

When a character string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded to the right with the necessary number of blanks. The pad character is always a blank even for strings defined with the FOR BIT DATA attribute.

Retrieval assignment of C NUL-terminated host variables is handled based on options specified with the PREP or BIND command. See the section on programming in C and C++ in the *Application Development Guide* for details.

Conversion Rules for String Assignments

A character string or graphic string assigned to a column or host variable is first converted, if necessary, to the code page of the target. Character conversion is necessary only if all of the following are true:

- The code pages are different.
- The string is neither null nor empty.
- Neither string has a code page value of 0 (FOR BIT DATA). [16]

MBCS Considerations for Character String Assignments

There are several considerations when assigning character strings that could contain both single and multi-byte characters. These considerations apply to all character strings, including those defined as FOR BIT DATA.

- Blank padding is always done using the single-byte blank character (X'20').

16. When acting as a DRDA application server, input host variables are converted to the code page of the application server, even if being assigned, compared or combined with a FOR BIT DATA column. If the SQLDA has been modified to identify the input host variable as FOR BIT DATA, conversion is not performed.

- Blank truncation is always done based on the single-byte blank character (X'20'). The double-byte blank character is treated as any other character with respect to truncation.
- Assignment of a character string to a host variable may result in fragmentation of MBCS characters if the target host variable is not large enough to contain the entire source string. If an MBCS character is fragmented, each byte of the MBCS character fragment in the target is set to a single-byte blank character (X'20'), no further bytes are moved from the source, and SQLWARN1 is set to ’W’ to indicate truncation. Note that the same MBCS character fragment handling applies even when the character string is defined as FOR BIT DATA.

DBCS Considerations for Graphic String Assignments

Graphic string assignments are processed in a manner analogous to that for character strings. Graphic string data types are compatible only with other graphic string data types, and never with numeric, character string, or datetime data types.

If a graphic string value is assigned to a graphic string column, the length of the value must not be greater than the length of the column.

If a graphic string value (the ’source’ string) is assigned to a fixed length graphic string data type (the ’target’, which can be a column or host variable), and the length of the source string is less than that of the target, the target will contain a copy of the source string which has been padded on the right with the necessary number of double-byte blank characters to create a value whose length equals that of the target.

If a graphic string value is assigned to a graphic string host variable and the length of the source string is greater than the length of the host variable, the host variable will contain a copy of the source string which has been truncated on the right by the necessary number of double-byte characters to create a value whose length equals that of the host variable. (Note that for this scenario, truncation need not be concerned with bisection of a double-byte character; if bisection were to occur, either the source value or target host variable would be an ill-defined graphic string data type.) The warning flag SQLWARN1 in the SQLCA will be set to ’W’. The indicator variable, if specified, will contain the original length (in double-byte characters) of the source string. In the case of DBCLOB, however, the indicator variable does not contain the original length.

Retrieval assignment of C NUL-terminated host variables (declared using wchar_t) is handled based on options specified with the PREP or BIND command. See the section on programming in C and C++ in the *Application Development Guide* for details.

Datetime Assignments

The basic rule for datetime assignments is that a DATE, TIME, or TIMESTAMP value may only be assigned to a column with a matching data type (whether DATE, TIME, or TIMESTAMP) or to a fixed− or varying−length character string variable or string column. The assignment must not be to a LONG VARCHAR, BLOB, or CLOB variable or column.

When a datetime value is assigned to a character string variable or string column, conversion to a string representation is automatic. Leading zeros are not omitted from any part of the date, time, or timestamp. The required length of the target will vary, depending on the format of the string representation. If the length of the target is greater than required, and the target is a fixed-length string, it is padded on the right with blanks. If the length of the target is less than required, the result depends on the type of datetime value involved, and on the type of target.

When the target is a host variable, the following rules apply:

- **For a DATE:** If the variable length is less than 10 bytes, an error occurs.
- **For a TIME:** If the USA format is used, the length of the variable must not be less than 8; in other formats the length must not be less than 5.

  If ISO or JIS formats are used, and if the length of the host variable is less than 8, the seconds part of the time is omitted from the result and assigned to the indicator variable, if provided. The SQLWARN1 field of the SQLCA is set to indicate the omission.
- **For a TIMESTAMP:** If the host variable is less than 19 bytes, an error occurs. If the length is less than 26, but greater than or equal to 19 bytes, trailing digits of the microseconds part of the value are omitted. The SQLWARN1 field of the SQLCA is set to indicate the omission.

For further information on string lengths for datetime values, see “Datetime Values” on page 82.

DATALINK Assignments

The assignment of a value to a DATALINK column results in the establishment of a link to a file unless the linkage attributes of the value are empty or the column is defined with NO LINK CONTROL. In cases where a linked value already exists in the column, that file is unlinked. Assigning a null value where a linked value already exists also unlinks the file associated with the old value.

If the application provides the same data location as already exists in the column, the link is retained. There are two reasons that this might be done:

- the comment is being changed

- if the table is placed in Datalink Reconcile Not Possible (DRNP) state, the links in the table can be reinstated by providing linkage attributes identical to the ones in the column.

A DATALINK value may be assigned to a column in any of the following ways:

- The DLVALUE scalar function can be used to create a new DATALINK value and assign it to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.
- A DATALINK value can be constructed in a CLI parameter using the CLI function SQLBuildDataLink. This value can then be assigned to a column. Unless the value contains only a comment or the URL is exactly the same, the act of assignment will link the file.

When assigning a value to a DATALINK column, the following error conditions return SQLSTATE 428D1:

- Data Location (URL) format is invalid (reason code 21).
- File server is not registered with this database (reason code 22).
- Invalid link type specified (reason code 23).
- Invalid length of comment or URL (reason code 27).

  Note that the size of a URL parameter or function result is the same on both input or output and is bound by the length of the DATALINK column. However, in some cases the URL value returned has an access token attached. In situations where this is possible, the output location must have sufficient storage space for the access token and the length of the DATALINK column. Hence, the actual length of the comment and URL in its fully expanded form, including any default URL scheme or default hostname, provided on input should be restricted to accomodate the output storage space. If the restricted length is exceeded, this error is raised.

When the assignment is also creating a link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File does not exist (SQLSTATE 428D1, reason code 24).
- Referenced file cannot be accessed for linking (reason code 26).
- File already linked to another column (SQLSTATE 428D1, reason code 25).

  Note that this error will be raised even if the link is to a different database.

In addition, when the assignment removes an existing link, the following errors can occur:

- File server not currently available (SQLSTATE 57050).
- File with referential integrity control is not in a correct state according to the Data Links File Manager (SQLSTATE 58004).

A DATALINK value may be retrieved from the database in either of the following ways:

- Portions of a DATALINK value can be assigned to host variables by use of scalar functions (such as DLLINKTYPE or DLURLPATH).

Note that usually no attempt is made to access the file server at retrieval time.[17] It is therefore possible that subsequent attempts to access the file server through file system commands might fail.

When retrieving a DATALINK, the registry of file servers at the database server is checked to confirm that the file server is still registered with the database server (SQLSTATE 55022). In addition, a warning may be returned when retrieving a DATALINK value because the table is in reconcile pending or reconcile not possible state (SQLSTATE 01627).

User-defined Type Assignments

With user-defined types, different rules are applied for assignments to host variables than are used for all other assignments.

Distinct Types: Assignment to host variables is done based on the source type of the distinct type. That is, it follows the rule:

- A value of a distinct type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the source type of this distinct type is assignable to this host variable.

If the target of the assignment is a column based on a distinct type, the source data type must be castable to the target data type as described in "Casting Between Data Types" on page 91 for user-defined types.

Structured Types: Assignment to and from host variables is based on the declared type of the host variable. That is, it follows the rule:

> A value of a structured type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the declared type of the host variable is the structured type or a supertype of the structured type.

If the target of the assignment is a column of a structured type, the source data type must be the target data type or a subtype of the target data type.

17. It may be necessary to access the file server to determine the prefix name associated with a path. This can be changed at the file server when the mount point of a file system is moved. First access of a file on a server will cause the required values to be retrieved from the file server and cached at the database server for the subsequent retrieval of DATALINK values for that file server. An error is returned if the file server cannot be accessed (SQLSTATE 57050).

Reference Type Assignments

A reference type with a target type of *T* can be assigned to a reference type column that is also a reference type with target type of *S* where *S* is a supertype of *T*. If an assignment is made to a scoped reference column or variable, no check is performed to ensure that the actual value being assigned exists in the target table or view defined by the scope.

Assignment to host variables is done based on the representation type of the reference type. That is, it follows the rule:

- A value of a reference type on the right hand side of an assignment is assignable to a host variable on the left hand side if and only if the representation type of this reference type is assignable to this host variable.

If the target of the assignment is a column, and the right hand side of the assignment is a host variable, the host variable must be explicitly cast to the reference type of the target column.

Numeric Comparisons

Numbers are compared algebraically; that is, with regard to sign. For example, −2 is less than +1.

If one number is an integer and the other is decimal, the comparison is made with a temporary copy of the integer, which has been converted to decimal.

When decimal numbers with different scales are compared, the comparison is made with a temporary copy of one of the numbers that has been extended with trailing zeros so that its fractional part has the same number of digits as the other number.

If one number is floating-point and the other is integer or decimal, the comparison is made with a temporary copy of the other number, which has been converted to double-precision floating-point.

Two floating-point numbers are equal only if the bit configurations of their normalized forms are identical.

String Comparisons

Character strings are compared according to the collating sequence specified when the database was created, except those with a FOR BIT DATA attribute which are always compared according to their bit values.

When comparing character strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with single-byte blanks sufficient to extend its length to that of the longer string. This logical extension is done for all character strings including those tagged as FOR BIT DATA.

Character strings (except character strings tagged as FOR BIT DATA) are compared according to the collating sequence specified when the database was created (see the *Administration Guide* for more information on collating sequences specified at database creation time). For example, the default collating sequence supplied by the database manager may give lowercase and uppercase versions of the same character the same weight. The database manager performs a two-pass comparison to ensure that only identical strings are considered equal to each other. In the first pass, strings are compared according to the database collating sequence. If the weights of the characters in the strings are equal, a second ″tie-breaker″ pass is performed to compare the strings on the basis of their actual code point values.

Two strings are equal if they are both empty or if all corresponding bytes are equal. If either operand is null, the result is unknown.

Long strings and LOB strings are not supported in any comparison operations that use the basic comparison operators (=, <>, <, >, <=, and >=). They are supported in comparisons using the LIKE predicate and the POSSTR function. See "LIKE Predicate" on page 197 and see "POSSTR" on page 336 for details.

Portions of long strings and LOB strings of up to 4 000 bytes can be compared using the SUBSTR and VARCHAR scalar functions. For example, given the columns:

```
MY_SHORT_CLOB   CLOB(300)
MY_LONG_VAR     LONG VARCHAR
```

then the following is valid:

```
WHERE VARCHAR(MY_SHORT_CLOB) > VARCHAR(SUBSTR(MY_LONG_VAR,1,300))
```

Examples:

For these examples, ′A′, ′Á′, ′a′, and ′á′, have the code point values X′41′, X′C1′, X′61′, and X′E1′ respectively.

Consider a collating sequence where the characters ′A′, ′Á′, ′a′, ′á′ have weights 136, 139, 135, and 138. Then the characters sort in the order of their weights as follows:

```
’a’ < ’A’ < ’á’ < ’Á’
```

Now consider four DBCS characters D1, D2, D3, and D4 with code points 0xC141, 0xC161, 0xE141, and 0xE161, respectively. If these DBCS characters are in CHAR columns, they sort as a sequence of bytes according to the collation weights of those bytes. First bytes have weights of 138 and 139, therefore D3 and D4 come before D2 and D1; second bytes have weights of 135 and 136. Hence, the order is as follows:

```
D4 < D3 < D2 < D1
```

However, if the values being compared have the FOR BIT DATA attribute, or if these DBCS characters were stored in a GRAPHIC column, the collation weights are ignored, and characters are compared according to their code points as follows:

```
'A' < 'a' < 'Á' < 'á'
```

The DBCS characters sort as sequence of bytes, in the order of code points as follows:

```
D1 < D2 < D3 < D4
```

Now consider a collating sequence where the characters 'A', 'Á', 'a', 'á' have (non-unique) weights 74, 75, 74, and 75. Considering collation weights alone (first pass), 'a' is equal to 'A', and 'á' is equal to 'Á'. The code points of the characters are used to break the tie (second pass) as follows:

```
'A' < 'a' < 'Á' < 'á'
```

DBCS characters in CHAR columns sort a sequence of bytes, according to their weights (first pass) and then according to their code points to break the tie (second pass). First bytes have equal weights, so the code points (0xC1 and 0xE1) break the tie. Therefore, characters D1 and D2 sort before characters D3 and D4. Then the second bytes are compared in similar way, and the final result is as follows:

```
D1 < D2 < D3 < D4
```

Once again, if the data in CHAR columns have the FOR BIT DATA attribute, or if the DBCS characters are stored in a GRAPHIC column, the collation weights are ignored, and characters are compared according to their code points:

```
D1 < D2 < D3 < D4
```

For this particular example, the result happens to be the same as when collation weights were used, but obviously this is not always the case.

Conversion Rules for Comparison

When two strings are compared, one of the strings is first converted, if necessary, to the code page set of the other string. For details, see “Rules for String Conversions” on page 111.

Ordering of Results

Results that require sorting are ordered based on the string comparison rules discussed in “String Comparisons” on page 102. The comparison is performed at the database server. On returning results to the client application, code page conversion may be performed. This subsequent code page conversion does not affect the order of the server-determined result set.

MBCS Considerations for String Comparisons

Mixed SBCS/MBCS character strings are compared according to the collating sequence specified when the database was created. For databases created with default (SYSTEM) collation sequence, all single-byte ASCII characters are sorted in correct order, but double-byte characters are not necessarily in code point sequence. For databases created with IDENTITY sequence, all double-byte characters are correctly sorted in their code point order, but single-byte ASCII characters are sorted in their code point order as well. For databases created with COMPATIBILITY sequence, a compromise order is used that sorts properly for most double-byte characters, and is almost correct for ASCII. This was the default collation table in DB2 Version 2.

Mixed character strings are compared byte-by-byte. This may result in unusual results for multi-byte characters that occur in mixed strings, because each byte is considered independently.

Example:

For this example, ’A’, ’B’, ’a’, and ’b’ double-byte characters have the code point values X'8260', X'8261', X'8281', and X'8282', respectively.

Consider a collating sequence where the code points X'8260', X'8261', X'8281', and X'8282' have weights 96, 65, 193, and 194. Then:

```
'B' < 'A' < 'a' < 'b'
```

and

```
'AB' < 'AA' < 'Aa' < 'Ab' < 'aB' < 'aA' < 'aa' < 'ab'
```

Graphic string comparisons are processed in a manner analogous to that for character strings.

Graphic string comparisons are valid between all graphic string data types except LONG VARGRAPHIC. LONG VARGRAPHIC and DBCLOB data types are not allowed in a comparison operation.

For graphic strings, the collating sequence of the database is not used. Instead, graphic strings are always compared based on the numeric (binary) values of their corresponding bytes.

Using the previous example, if the literals were graphic strings, then:

```
'A' < 'B' < 'a' < 'b'
```

and

```
'AA' < 'AB' < 'Aa' < 'Ab' < 'aA' < 'aB' < 'aa' < 'ab'
```

When comparing graphic strings of unequal lengths, the comparison is made using a logical copy of the shorter string which is padded on the right with double-byte blank characters sufficient to extend its length to that of the longer string.

Two graphic values are equal if they are both empty or if all corresponding graphics are equal. If either operand is null, the result is unknown. If two values are not equal, their relation is determined by a simple binary string comparison.

As indicated in this section, comparing strings on a byte by byte basis can produce unusual results; that is, a result that differs from what would be expected in a character by character comparison. The examples shown here assume the same MBCS code page, however, the situation can be further complicated when using different multi-byte code pages with the same national language. For example, consider the case of comparing a string from a Japanese DBCS code page and a Japanese EUC code page.

Datetime Comparisons

A DATE, TIME, or TIMESTAMP value may be compared either with another value of the same data type or with a string representation of that data type. All comparisons are chronological, which means the farther a point in time is from January 1, 0001, the greater the value of that point in time.

Comparisons involving TIME values and string representations of time values always include seconds. If the string representation omits seconds, zero seconds is implied.

Comparisons involving TIMESTAMP values are chronological without regard to representations that might be considered equivalent.

Example:

```
TIMESTAMP('1990-02-23-00.00.00') > '1990-02-22-24.00.00'
```

User-defined Type Comparisons

Values with a user-defined distinct type can only be compared with values of exactly the same user-defined distinct type. The user-defined distinct type must have been defined using the WITH COMPARISONS clause.

Example:

Given the following YOUTH distinct type and CAMP_DB2_ROSTER table:

```
CREATE DISTINCT TYPE YOUTH AS INTEGER WITH COMPARISONS

CREATE TABLE CAMP_DB2_ROSTER
```

```
( NAME                 VARCHAR(20),
  ATTENDEE_NUMBER      INTEGER NOT NULL,
  AGE                  YOUTH,
  HIGH_SCHOOL_LEVEL    YOUTH)
```

The following comparison is valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > HIGH_SCHOOL_LEVEL
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > ATTENDEE_NUMBER
```

However, AGE can be compared to ATTENDEE_NUMBER by using a function or CAST specification to cast between the distinct type and the source type. The following comparisons are all valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE INTEGER(AGE) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
   WHERE CAST( AGE AS INTEGER) > ATTENDEE_NUMBER

SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > YOUTH(ATTENDEE_NUMBER)

SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > CAST(ATTENDEE_NUMBER AS YOUTH)
```

Values with a user-defined structured type cannot be compared with any other value (the NULL predicate and the TYPE predicate can be used).

Reference Type Comparisons

Reference type values can be compared only if their target types have a common supertype. The appropriate comparison function will only be found if the schema name of the common supertype is included in the function path. The comparison is performed using the representation type of the reference types. The scope of the reference is not considered in the comparison.

Rules for Result Data Types

The data types of a result are determined by rules which are applied to the operands in an operation. This section explains those rules.

These rules apply to:

- Corresponding columns in fullselects of set operations (UNION, INTERSECT and EXCEPT)
- Result expressions of a CASE expression
- Arguments of the scalar function COALESCE (or VALUE)

- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

These rules are applied subject to other restrictions on long strings for the various operations.

The rules involving various data types follow. In some cases, a table is used to show the possible result data types.

These tables identify the data type of the result, including the applicable length or precision and scale. The result type is determined by considering the operands. If there is more than one pair of operands, start by considering the first pair. This gives a result type which is considered with the next operand to determine the next result type, and so on. The last intermediate result type and the last operand determine the result type for the operation. Processing of operations is done from left to right so that the intermediate result types are important when operations are repeated. For example, consider a situation involving:

```
CHAR(2) UNION CHAR(4) UNION VARCHAR(3)
```

The first pair results in a type of CHAR(4). The result values always have 4 characters. The final result type is VARCHAR(4). Values in the result from the first UNION operation will always have a length of 4.

Character Strings

Character strings are compatible with other character strings. Character strings include data types CHAR, VARCHAR, LONG VARCHAR, and CLOB.

If one operand is...	And the other operand is...	The data type of the result is...
CHAR(x)	CHAR(y)	CHAR(z) where z = max(x,y)
CHAR(x)	VARCHAR(y)	VARCHAR(z) where z = max(x,y)
VARCHAR(x)	CHAR(y) or VARCHAR(y)	VARCHAR(z) where z = max(x,y)
LONG VARCHAR	CHAR(y), VARCHAR(y), or LONG VARCHAR	LONG VARCHAR
CLOB(x)	CHAR(y), VARCHAR(y), or CLOB(y)	CLOB(z) where z = max(x,y)
CLOB(x)	LONG VARCHAR	CLOB(z) where z = max(x,32700)

The code page of the result character string will be derived based on the “Rules for String Conversions” on page 111.

Graphic Strings

Graphic strings are compatible with other graphic strings. Graphic strings include data types GRAPHIC, VARGRAPHIC, LONG VARGRAPHIC, and DBCLOB.

If one operand is...	And the other operand is...	The data type of the result is...
GRAPHIC(x)	GRAPHIC(y)	GRAPHIC(z) where z = max(x,y)
VARGRAPHIC(x)	GRAPHIC(y) OR VARGRAPHIC(y)	VARGRAPHIC(z) where z = max(x,y)
LONG VARGRAPHIC	GRAPHIC(y), VARGRAPHIC(y), or LONG VARGRAPHIC	LONG VARGRAPHIC
DBCLOB(x)	GRAPHIC(y), VARGRAPHIC(y), or DBCLOB(y)	DBCLOB(z) where z = max (x,y)
DBCLOB(x)	LONG VARGRAPHIC	DBCLOB(z) where z = max (x,16350)

The code page of the result graphic string will be derived based on the "Rules for String Conversions" on page 111.

Binary Large Object (BLOB)

A BLOB is compatible only with another BLOB and the result is a BLOB. The BLOB scalar function should be used to cast from other types if they should be treated as BLOB types (see "BLOB" on page 258). The length of the result BLOB is the largest length of all the data types.

Numeric

Numeric types are compatible with other numeric types. Numeric types include SMALLINT, INTEGER, BIGINT, DECIMAL, REAL and DOUBLE.

If one operand is...	And the other operand is...	The data type of the result is...
SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER
INTEGER	SMALLINT	INTEGER
BIGINT	BIGINT	BIGINT
BIGINT	INTEGER	BIGINT
BIGINT	SMALLINT	BIGINT
DECIMAL(w,x)	SMALLINT	DECIMAL(p,x) where p = x+max(w-x,5)[1]

If one operand is...	And the other operand is...	The data type of the result is...
DECIMAL(w,x)	INTEGER	DECIMAL(p,x) where $p = x+\max(w-x,11)$[1]
DECIMAL(w,x)	BIGINT	DECIMAL(p,x) where $p = x+\max(w-x,19)$[1]
DECIMAL(w,x)	DECIMAL(y,z)	DECIMAL(p,s) where $p = \max(x,z)+\max(w-x,y-z)$[1]$s = \max(x,z)$
REAL	REAL	REAL
REAL	DECIMAL, BIGINT, INTEGER, or SMALLINT	DOUBLE
DOUBLE	any numeric	DOUBLE

Note: 1. Precision cannot exceed 31.

DATE

A date is compatible with another date, or any CHAR or VARCHAR expression that contains a valid string representation of a date. The data type of the result is DATE.

TIME

A time is compatible with another time, or any CHAR or VARCHAR expression that contains a valid string representation of a time. The data type of the result is TIME.

TIMESTAMP

A timestamp is compatible with another timestamp, or any CHAR or VARCHAR expression that contains a valid string representation of a timestamp. The data type of the result is TIMESTAMP.

DATALINK

A datalink is compatible with another datalink. The data type of the result is DATALINK. The length of the result DATALINK is the largest length of all the data types.

User-defined Types

Distinct Types

A user-defined distinct type is compatible only with the same user-defined distinct type. The data type of the result is the user-defined distinct type.

Reference Types

A reference type is compatible with another reference type provided that their target types have a common supertype. The data type of the result is a reference type having the common supertype as the target type. If all operands have the identical scope table, the result has that scope table. Otherwise the result is unscoped.

Structured Types

A structured type is compatible with another structured type provided that they have a common supertype. The static data type of the resulting structured type column is the structured type that is the least common supertype of either column.

For example, consider the following structured type hierarchy,

```
      A
     / \
    B   C
   / \
  D   E
 / \
F   G
```

Structured types of the static type E and F are compatible with the resulting static type of B, which is the least common super type of E and F.

Nullable Attribute of Result

With the exception of INTERSECT and EXCEPT, the result allows nulls unless both operands do not allow nulls.

- For INTERSECT, if either operand does not allow nulls the result does not allow nulls (the intersection would never be null).
- For EXCEPT, if the first operand does not allow nulls the result does not allow nulls (the result can only be values from the first operand).

Rules for String Conversions

The code page used to perform an operation is determined by rules which are applied to the operands in that operation. This section explains those rules.

These rules apply to:

- Corresponding string columns in fullselects with set operations (UNION, INTERSECT and EXCEPT)
- Operands of concatenation
- Operands of predicates (with the exception of LIKE)
- Result expressions of a CASE expression
- Arguments of the scalar function COALESCE (and VALUE)
- Expression values of the in list of an IN predicate
- Corresponding expressions of a multiple row VALUES clause.

In each case, the code page of the result is determined at bind time, and the execution of the operation may involve conversion of strings to the code page identified by that code page. A character that has no valid conversion is mapped to the substitution character for the character set and SQLWARN10 is set to ’W’ in the SQLCA.

The code page of the result is determined by the code pages of the operands. The code pages of the first two operands determine an intermediate result code page, this code page and the code page of the next operand determine a new intermediate result code page (if applicable), and so on. The last intermediate result code page and the code page of the last operand determine the code page of the result string or column. For each pair of code pages, the result is determined by the sequential application of the following rules:

- If the code pages are equal, the result is that code page.
- If either code page is BIT DATA (code page 0), the result code page is BIT DATA.
- Otherwise, the result code page is determined by Table 8. An entry of ’first’ in the table means the code page from the first operand is selected and an entry of ’second’ means the code page from the second operand is selected.

Table 8. Selecting the Code Page of the Intermediate Result

	Second Operand				
First Operand	**Column Value**	**Derived Value**	**Constant**	**Special Register**	**Host Variable**
Column Value	first	first	first	first	first
Derived Value	second	first	first	first	first
Constant	second	second	first	first	first
Special Register	second	second	first	first	first
Host Variable	second	second	second	second	first

An intermediate result is considered to be a derived value operand. An expression that is not a single column value, constant, special register, or host variable is also considered a derived value operand. There is an exception to this if the expression is a CAST specification (or a call to a function that is equivalent). In this case, the *kind* for the first operand is based on the first argument of the CAST specification.

View columns are considered to have the operand type of the object on which they are ultimately based. For example, a view column defined on a table column is considered to be a column value, whereas a view column based on a string expression (for example, A CONCAT B) is considered to be a derived value.

Conversions to the code page of the result are performed, if necessary, for:
- An operand of the concatenation operator
- The selected argument of the COALESCE (or VALUE) scalar function
- The selected result expression of the CASE expression
- The expressions of the in list of the IN predicate
- The corresponding expressions of a multiple row VALUES clause
- The corresponding columns involved in set operations.

Character conversion is necessary if all of the following are true:
- The code pages are different
- Neither string is BIT DATA
- The string is neither null nor empty
- The code page conversion selection table indicates that conversion is necessary.

Examples

Example 1: Given the following:

Expression	Type	Code Page
COL_1	column	850
HV_2	host variable	437

When evaluating the predicate:

```
COL_1 CONCAT :HV_2
```

The result code page of the two operands is 850, since the dominant operand is the column COL_1.

Example 2: Using the information from the previous example, when evaluating the predicate:

```
COALESCE(COL_1, :HV_2:NULLIND,)
```

The result code page is 850. Therefore the result code page for the COALESCE scalar function will be the code page 850.

Partition Compatibility

Partition compatibility is defined between the base data types of corresponding columns of partitioning keys. Partition compatible data types have the property that two variables, one of each type, with the same value, are mapped to the same partitioning map index by the same partitioning function.

Table 9 shows the compatibility of data types in partitions.

Partition compatibility has the following characteristics:

- Internal formats are used for DATE, TIME, and TIMESTAMP. They are not compatible with each other, and none are compatible with CHAR.
- Partition compatibility is not affected by columns with NOT NULL or FOR BIT DATA definitions.
- NULL values of compatible data types are treated identically. Different results might be produced for NULL values of non-compatible data types.
- Base datatype of the UDT is used to analyze partition compatibility.
- Decimals of the same value in the partitioning key are treated identically, even if their scale and precision differ.
- Trailing blanks in character strings (CHAR, VARCHAR, GRAPHIC or VARGRAPHIC) are ignored by the system-provided hashing function.
- CHAR or VARCHAR of different lengths are compatible data types.
- REAL or DOUBLE values that are equal are treated identically even though their precision differs.

Table 9. Partition Compatibilities

Operands	**Binary Integer**	**Decimal Number**	**Floating Point**	**Character String**	**Graphic String**	**Date**	**Time**	**Time-stamp**	**Distinct Type**	**Structured Type**
Binary Integer	Yes	No	No	No	No	No	No	No	^{1}	No
Decimal Number	No	Yes	No	No	No	No	No	No	^{1}	No
Floating Point	No	No	Yes	No	No	No	No	No	^{1}	No
Character String^3	No	No	No	Yes^2	No	No	No	No	^{1}	No
Graphic String^3	No	No	No	No	Yes	No	No	No	^{1}	No
Date	No	No	No	No	No	Yes	No	No	^{1}	No
Time	No	No	No	No	No	No	Yes	No	^{1}	No
Timestamp	No	No	No	No	No	No	No	Yes	^{1}	No

Table 9. Partition Compatibilities (continued)

Operands	Binary Integer	Decimal Number	Floating Point	Character String	Graphic String	Date	Time	Time-stamp	Distinct Type	Structured Type
Distinct Type	[1]	[1]	[1]	[1]	[1]	[1]	[1]	[1]	[1]	No
Structured Type[3]	No	No	No	No	No	No	No	No	No	No

Note:

[1] A user-defined distinct type (UDT) value is partition compatible with the source type of the UDT or any other UDT with a partition compatible source type.

[2] The FOR BIT DATA attribute does not affect the partition compatibility.

[3] Note that user-defined structured types and data types LONG VARCHAR, LONG VARGRAPHIC, CLOB, DBCLOB, and BLOB are not applicable for partition compatibility since they are not supported in partitioning keys.

Constants

A *constant* (sometimes called a *literal*) specifies a value. Constants are classified as string constants or numeric constants. Numeric constants are further classified as integer, floating-point, or decimal.

All constants have the attribute NOT NULL.

A negative zero value in a numeric constant (-0) is the same value as a zero without the sign (0).

Integer Constants

An *integer constant* specifies an integer as a signed or unsigned number with a maximum of 19 digits that does not include a decimal point. The data type of an integer constant is a large integer if its value is within the range of a large integer. The data type of an integer constant is big integer if its value is outside the range of large integer but within the range of a big integer. A constant that is defined outside the range of big integer values is considered a decimal constant.

Note that the smallest literal representation of a large integer constant is -2 147 483 647 and not -2 147 483 648, which is the limit for integer values. Similarly, the smallest literal representation of a big integer constant is -9 223 372 036 854 775 807 and not -9 223 372 036 854 775 808 which is the limit for big integer values.

Examples

```
64     -15     +100     32767     720176      12345678901
```

In syntax diagrams the term 'integer' is used for a large integer constant that must not include a sign.

Floating-Point Constants

A *floating-point constant* specifies a floating-point number as two numbers separated by an E. The first number may include a sign and a decimal point; the second number may include a sign but not a decimal point. The data type of a floating-point constant is double precision. The value of the constant is the product of the first number and the power of 10 specified by the second number; it must be within the range of floating-point numbers. The number of characters in the constant must not exceed 30.

Examples

```
15E1     2.E5     2.2E-1     +5.E+2
```

Decimal Constants

A *decimal constant* is a signed or unsigned number that consists of no more than 31 digits and either includes a decimal point or is not within the range of binary integers. It must be in the range of decimal numbers. The precision is the total number of digits (including leading and trailing zeros); the scale is the number of digits to the right of the decimal point (including trailing zeros).

Examples

```
25.5     1000.     -15.     +37589.3333333333
```

Character String Constants

A *character string constant* specifies a varying-length character string and consists of a sequence of characters that starts and ends with an apostrophe ('). This form of string constant specifies the character string contained between the string delimiters. The length of the character string must not be greater than 32 672 bytes. Two consecutive string delimiters are used to represent one string delimiter within the character string.

Examples

```
   '12/14/1985'
   '32'
   'DON''T CHANGE'
```

Unequal Code Page Considerations

The constant value is always converted to the database code page when it is bound to the database. It is considered to be in the database code page. Therefore, if used in an expression that combines a constant with a FOR BIT DATA column, of which the result is FOR BIT DATA, the constant value will *not* be converted from its database code page representation when used.

Hexadecimal Constants

A *hexadecimal constant* specifies a varying-length character string with the code page of the application server.

The format of a hexadecimal string constant is an X followed by a sequence of characters that starts and ends with an apostrophe (single quote). The characters between the apostrophes must be an even number of hexadecimal digits. The number of hexadecimal digits must not exceed 16 336, otherwise an error is raised (SQLSTATE -54002). A hexadecimal digit represents 4 bits. It is specified as a digit or any of the letters A through F (uppercase or lowercase) where A represents the bit pattern ’1010’, B the bit pattern ’1011’, etc. If a hexadecimal constant is improperly formatted (e.g. it contains an invalid hexadecimal digit or an odd number of hexadecimal digits), an error is raised (SQLSTATE 42606).

Examples

```
X'FFFF'       representing the bit pattern '1111111111111111'

X'4672616E6B' representing the VARCHAR pattern of the ASCII string 'Frank'
```

Graphic String Constants

A *graphic string constant* specifies a varying-length graphic string and consists of a sequence of double-byte characters that starts and ends with a single-byte apostrophe (') and is preceded by a single-byte G or N. This form of string constant specifies the graphic string contained between the string delimiters. The length of the graphic string must be an even number of bytes and must not be greater than 16 336 bytes.

Examples:

```
G'double-byte character string'
N'double-byte character string'
```

MBCS Considerations

The apostrophe must not appear as part of an MBCS character to be considered a delimiter.

Using Constants with User-defined Types

User-defined types have strong typing. This means that a user-defined type is only compatible with its own type. A constant, however, has a built-in type. Therefore, an operation involving a user-defined type and a constant is only possible if the user-defined type has been cast to the constant’s built-in type or the constant has been cast to the user-defined type (see “CAST Specifications” on page 173 for information on casting). For example, using the table and distinct type in “User-defined Type Comparisons” on page 106, the following comparisons with the constant 14 are valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > CAST(14 AS YOUTH)
```

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE CAST(AGE AS INTEGER) > 14
```

The following comparison is not valid:

```
SELECT * FROM CAMP_DB2_ROSTER
   WHERE AGE > 14
```

Special Registers

A *special register* is a storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. Special registers are in the database code page.

CURRENT DATE

The CURRENT DATE special register specifies a date that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT TIME or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

In a federated system, CURRENT DATE can be used in a query intended for data sources. When the query is processed, the date returned will be obtained from the CURRENT DATE register at the federated server, not from the data sources.

Example

Using the PROJECT table, set the project end date (PRENDATE) of the MA2111 project (PROJNO) to the current date.

```
UPDATE PROJECT
  SET PRENDATE = CURRENT DATE
  WHERE PROJNO = 'MA2111'
```

CURRENT DEFAULT TRANSFORM GROUP

The CURRENT DEFAULT TRANSFORM GROUP special register specifies a VARCHAR (18) value that identifies the name of the transform group used by dynamic SQL statements for exchanging user-defined structured type values with host programs. This special register does not specify the transform groups used in static SQL statements or in the exchange of parameters and results with external functions of methods.

Its value can be set by the SET CURRENT DEFAULT TRANSFORM GROUP statement. If no value is set, the initial value of the special register is the empty string (a VARCHAR with a zero length).

In a dynamic SQL statement (that is, one which interacts with host variables), the name of the transform group used for exchanging values is the same as the value of this special register, unless this register contains the empty string. If the register contains the empty string (no value was set by using a SET CURRENT DEFAULT TRANSFORM GROUP statement), the DB2_PROGRAM transform group is used for the transform. If the DB2_PROGRAM transform group is not defined for the structured type subject, an error is raised at run time (SQLSTATE 42741).

Example

Set the default transform group to MYSTRUCT1. The TO SQL and FROM SQL functions defined in the MYSTRUCT1 transform are used to exchange user-defined structured type variables with the host program.

```
   SET CURRENT DEFAULT TRANSFORM GROUP = MYSTRUCT1
```

Retrieve the name of the default transform group assigned to this special register.

```
   VALUES (CURRENT DEFAULT TRANSFORM GROUP)
```

CURRENT DEGREE

The CURRENT DEGREE special register specifies the degree of intra-partition parallelism for the execution of dynamic SQL statements. [18] The data type of the register is CHAR(5). Valid values are ′ANY ′ or the string representation of an integer between 1 and 32 767, inclusive.

If the value of CURRENT DEGREE represented as an integer is 1 when an SQL statement is dynamically prepared, the execution of that statement will not use intra-partition parallelism.

If the value of CURRENT DEGREE represented as an integer is greater than 1 and less than or equal to 32 767 when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism with the specified degree.

If the value of CURRENT DEGREE is ′ANY′ when an SQL statement is dynamically prepared, the execution of that statement can involve intra-partition parallelism using a degree determined by the database manager.

The actual runtime degree of parallelism will be the lower of:

- Maximum query degree (max_querydegree) configuration parameter
- Application runtime degree
- SQL statement compilation degree

18. For static SQL, the DEGREE bind option provides the same control.

If the intra_parallel database manager configuration parameter is set to NO, the value of the CURRENT DEGREE special register will be ignored for the purpose of optimization, and the statement will not use intra-partition parallelism.

See the *Administration Guide* for a description of parallelism and a list of restrictions.

The value can be changed by executing the SET CURRENT DEGREE statement (see "SET CURRENT DEGREE" on page 1004 for information on this statement).

The initial value of CURRENT DEGREE is determined by the dft_degree database configuration parameter. See the *Administration Guide* for a description of this configuration parameter.

CURRENT EXPLAIN MODE

The CURRENT EXPLAIN MODE special register holds a VARCHAR(254) value which controls the behavior of the Explain facility with respect to eligible dynamic SQL statements. This facility generates and inserts Explain information into the Explain tables (for more information see the *Administration Guide*). This information does not include the Explain snapshot.

The possible values are YES, NO, EXPLAIN, RECOMMEND INDEXES and EVALUATE INDEXES. [19]

YES Enables the explain facility and causes explain information for a dynamic SQL statement to be captured when the statement is compiled.

EXPLAIN
Enables the facility like YES, however, the dynamic statements are not executed.

NO Disables the Explain facility.

RECOMMEND INDEXES
For each dynamic query, a set of indexes is recommended. ADVISE_INDEX table is populated with the set of indexes.

EVALUATE INDEXES
Dynamic queries are explained as if the recommended indexes existed. The indexes are picked up from ADVISE_INDEX table.

The initial value is NO.

19. For static SQL, the EXPLAIN bind option provides the same control. In the case of the PREP and BIND commands, the EXPLAIN option values are: YES, NO and ALL.

Its value can be changed by the SET CURRENT EXPLAIN MODE statement (see "SET CURRENT EXPLAIN MODE" on page 1006 for information on this statement).

The CURRENT EXPLAIN MODE and CURRENT EXPLAIN SNAPSHOT special register values interact when the Explain facility is invoked (see Table 142 on page 1325 for details). The CURRENT EXPLAIN MODE special register also interacts with the EXPLAIN bind option (see Table 143 on page 1326 for details). The values RECOMMEND INDEXES and EVALUATE INDEXES can only be set for the CURRENT EXPLAIN MODE register, and must be set using the SET CURRENT EXPLAIN MODE statement.

Example: Set the host variable EXPL_MODE (VARCHAR(254)) to the value currently in the CURRENT EXPLAIN MODE special register.

```
VALUES CURRENT EXPLAIN MODE
  INTO :EXPL_MODE
```

CURRENT EXPLAIN SNAPSHOT

The CURRENT EXPLAIN SNAPSHOT special register holds a CHAR(8) value which controls the behavior of the Explain snapshot facility. This facility generates compressed information including access plan information, operator costs, and bind-time statistics (for more information see the *Administration Guide*).

Only the following statements consider the value of this register: DELETE, INSERT, SELECT, SELECT INTO, UPDATE, VALUES or VALUES INTO.

The possible values are YES, NO, and EXPLAIN. [20]

YES Enables the snapshot facility and takes a snapshot of the internal representation of a dynamic SQL statement as the statement is compiled.

EXPLAIN
Enables the facility like YES, however, the dynamic statements are not executed.

NO Disables the Explain snapshot facility.

The initial value is NO.

Its value can be changed by the SET CURRENT EXPLAIN SNAPSHOT statement (see "SET CURRENT EXPLAIN SNAPSHOT" on page 1008).

20. For static SQL, the EXPLSNAP bind option provides the same control. In the case of the PREP and BIND commands, the EXPLSNAP option values are: YES, NO and ALL.

The CURRENT EXPLAIN SNAPSHOT and CURRENT EXPLAIN MODE special register values interact when the Explain facility is invoked (see Table 142 on page 1325 for details). The CURRENT EXPLAIN SNAPSHOT special register also interacts with the EXPLSNAP bind option (see Table 144 on page 1327 for details).

Example

Set the host variable EXPL_SNAP (char(8)) to the value currently in the CURRENT EXPLAIN SNAPSHOT special register.

```
VALUES CURRENT EXPLAIN SNAPSHOT
  INTO :EXPL_SNAP
```

CURRENT NODE

The CURRENT NODE special register specifies an INTEGER value that identifies the coordinator node number (the partition to which an application connects).

CURRENT NODE returns 0 if the database instance is not defined to support partitioning (no db2nodes.cfg file[21]).

The CURRENT NODE can be changed by the CONNECT statement, but only under certain conditions (see "CONNECT (Type 1)" on page 550).

Example

Set the host variable APPL_NODE (integer) to the number of the partition to which the application is connected.

```
VALUES CURRENT NODE
  INTO :APPL_NODE
```

CURRENT PATH

The CURRENT PATH special register specifies a VARCHAR(254) value that identifies the *SQL path* to be used to resolve function references and data type references that are used in dynamically prepared SQL statements.[22] CURRENT PATH is also used to resolve stored procedure references in CALL statements. The initial value is the default value specified below. For static SQL, the FUNCPATH bind option provides a SQL path that is used for function and data type resolution (see the *Command Reference* for more information on the FUNCPATH bind option).

21. For partitioned databases, the db2nodes.cfg file exists and contains partition (or node) definitions. For details refer to the *Administration Guide*.

22. CURRENT FUNCTION PATH is a synonym for CURRENT PATH.

The CURRENT PATH special register contains a list of one or more schema-names, where the schema-names are enclosed in double quotes and separated by commas (any quotes within the string are repeated as they are in any delimited identifier).

For example, a SQL path specifying that the database manager is to first look in the FERMAT, then XGRAPHIC, then SYSIBM schemas is returned in the CURRENT PATH special register as:

```
"FERMAT","XGRAPHIC","SYSIBM"
```

The default value is ″SYSIBM″,″SYSFUN″,X where X is the value of the USER special register delimited by double quotes.

Its value can be changed by the SET CURRENT FUNCTION PATH statement (see “SET PATH” on page 1031). The schema SYSIBM does not need to be specified. If it is not included in the SQL path, it is implicitly assumed as the first schema. SYSIBM does not take any of the 254 characters if it is implicitly assumed.

The use of the SQL path for function resolution is described in “Functions” on page 142. A data type that is not qualified with a schema name will be implicitly qualified with the schema name that is earliest in the SQL path and contains a data type with the same unqualified name specified. There are exceptions to this rule as described in the following statements: CREATE DISTINCT TYPE, CREATE FUNCTION, COMMENT ON and DROP.

Example

Using the SYSCAT.VIEWS catalog view, find all views that were created with the exact same setting as the current value of the CURRENT PATH special register.

```
SELECT VIEWNAME, VIEWSCHEMA FROM SYSCAT.VIEWS
  WHERE FUNC_PATH = CURRENT PATH
```

CURRENT QUERY OPTIMIZATION

The CURRENT QUERY OPTIMIZATION special register specifies an INTEGER value that controls the class of query optimization performed by the database manager when binding dynamic SQL statements. The QUERYOPT bind option controls the class of query optimization for static SQL statements (see the *Command Reference* for additional information on the QUERYOPT bind option). The possible values range from 0 to 9. For example, if the query optimization class is set to the minimal class of optimization (0), then the value in the special register is 0. The default value is determined by the dft_queryopt database configuration parameter.

Its value can be changed by the SET CURRENT QUERY OPTIMIZATION statement (see “SET CURRENT QUERY OPTIMIZATION” on page 1012).

Example

Using the SYSCAT.PACKAGES catalog view, find all plans that were bound with the same setting as the current value of the CURRENT QUERY OPTIMIZATION special register.

```
SELECT PKGNAME, PKGSCHEMA FROM SYSCAT.PACKAGES
  WHERE QUERYOPT = CURRENT QUERY OPTIMIZATION
```

CURRENT REFRESH AGE

The CURRENT REFRESH AGE special register specifies a timestamp duration value with a data type of DECIMAL(20,6). This duration is the maximum duration since a REFRESH TABLE statement has been processed on a REFRESH DEFERRED summary table such that the summary table can be used to optimize the processing of a query. If CURRENT REFRESH AGE has a value of 99 999 999 999 999 (ANY), and QUERY OPTIMIZATION class is 5 or more, REFRESH DEFERRED summary tables are considered to optimize the processing of a dynamic SQL query. A summary table with the REFRESH IMMEDIATE attribute and not in check pending state is assumed to have a refresh age of zero.

Its value can be changed by the SET CURRENT REFRESH AGE statement (see “SET CURRENT REFRESH AGE” on page 1015). Summary tables defined with REFRESH DEFERRED are never considered by static embedded SQL queries.

The initial value of CURRENT REFRESH AGE is zero.

CURRENT SCHEMA

The CURRENT SCHEMA special register specifies a VARCHAR(128) value that identifies the schema name used to qualify unqualified database object references where applicable in dynamically prepared SQL statements. [23]

The initial value of CURRENT SCHEMA is the authorization ID of the current session user.

Its value can be changed by the SET SCHEMA statement (see “SET SCHEMA” on page 1033).

The QUALIFIER bind option controls the schema name used to qualify unqualified database object references where applicable for static SQL statements (see *Command Reference* for more information).

Example

Set the schema for object qualification to ’D123’.

```
SET CURRENT SCHEMA =  'D123'
```

23. For compatibility with DB2 for OS/390, the special register CURRENT SQLID is treated as a synonym for CURRENT SCHEMA.

CURRENT SERVER

The CURRENT SERVER special register specifies a VARCHAR(18) value that identifies the current application server. The actual name of the application server (not an alias) is contained in the register.

The CURRENT SERVER can be changed by the CONNECT statement, but only under certain conditions (see "CONNECT (Type 1)" on page 550).

Example

Set the host variable APPL_SERVE (VARCHAR(18)) to the name of the application server to which the application is connected.

```
VALUES CURRENT SERVER
  INTO :APPL_SERVE
```

CURRENT TIME

The CURRENT TIME special register specifies a time that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIMESTAMP within a single statement, all values are based on a single clock reading.

In a federated system, CURRENT TIME can be used in a query intended for data sources. When the query is processed, the time returned will be obtained from the CURRENT TIME register at the federated server, not from the data sources.

Example

Using the CL_SCHED table, select all the classes (CLASS_CODE) that start (STARTING) later today. Today's classes have a value of 3 in the DAY column.

```
SELECT CLASS_CODE FROM CL_SCHED
  WHERE STARTING > CURRENT TIME  AND DAY = 3
```

CURRENT TIMESTAMP

The CURRENT TIMESTAMP special register specifies a timestamp that is based on a reading of the time-of-day clock when the SQL statement is executed at the application server. If this special register is used more than once within a single SQL statement, or used with CURRENT DATE or CURRENT TIME within a single statement, all values are based on a single clock reading.

In a federated system, CURRENT TIMESTAMP can be used in a query intended for data sources. When the query is processed, the timestamp returned will be obtained from the CURRENT TIMESTAMP register at the federated server, not from the data sources.

Example
Insert a row into the IN_TRAY table. The value of the RECEIVED column should be a timestamp that indicates when the row was inserted. The values for the other three columns come from the host variables SRC (char(8)), SUB (char(64)), and TXT (VARCHAR(200)).

```
INSERT INTO IN_TRAY
  VALUES (CURRENT TIMESTAMP, :SRC, :SUB, :TXT)
```

CURRENT TIMEZONE

The CURRENT TIMEZONE special register specifies the difference between UTC [24] and local time at the application server. The difference is represented by a time duration (a decimal number in which the first two digits are the number of hours, the next two digits are the number of minutes, and the last two digits are the number of seconds). The number of hours is between -24 and 24 exclusive. Subtracting CURRENT TIMEZONE from a local time converts that local time to UTC. The time is calculated from the operating system time at the moment the SQL statement is executed. [25]

The CURRENT TIMEZONE special register can be used wherever an expression of the DECIMAL(6,0) data type is used, for example, in time and timestamp arithmetic.

Example
Insert a record into the IN_TRAY table, using a UTC timestamp for the RECEIVED column.

```
INSERT INTO IN_TRAY VALUES (
       CURRENT TIMESTAMP - CURRENT TIMEZONE,
       :source,
       :subject,
       :notetext )
```

USER

The USER special register specifies the run-time authorization ID passed to the database manager when an application starts on a database. The data type of the register is VARCHAR(128).

Example
Select all notes from the `IN_TRAY` table that the user placed there himself.

```
SELECT * FROM IN_TRAY
  WHERE SOURCE = USER
```

24. Coordinated Universal Time, formerly known as GMT.

25. The CURRENT TIMEZONE value is determined from C runtime functions. See the *Quick Beginnings* for any installation requirements regarding time zone.

Column Names

The meaning of a *column name* depends on its context. A column name can be used to:

- Declare the name of a column, as in a CREATE TABLE statement.
- Identify a column, as in a CREATE INDEX statement.
- Specify values of the column, as in the following contexts:
  - In a column function, a column name specifies all values of the column in the group or intermediate result table to which the function is applied. (Groups and intermediate result tables are explained under "Chapter 5. Queries" on page 393.) For example, MAX(SALARY) applies the function MAX to all values of the column SALARY in a group.
  - In a GROUP BY or ORDER BY clause, a column name specifies all values in the intermediate result table to which the clause is applied. For example, ORDER BY DEPT orders an intermediate result table by the values of the column DEPT.
  - In an expression, a search condition, or a scalar function, a column name specifies a value for each row or group to which the construct is applied. For example, when the search condition CODE = 20 is applied to some row, the value specified by the column name CODE is the value of the column CODE in that row.
- Temporarily rename a column, as in the *correlation-clause* of a *table-reference* in a FROM clause.

Qualified Column Names

A qualifier for a column name may be a table, view, nickname, alias, or correlation name.

Whether a column name may be qualified depends on its context:

- Depending on the form of the COMMENT ON statement, a single column name may need to be qualified. Multiple column names must be unqualified.
- Where the column name specifies values of the column, it may be qualified at the user's option.
- In all other contexts, a column name must not be qualified.

Where a qualifier is optional, it can serve two purposes. They are described under "Column Name Qualifiers to Avoid Ambiguity" on page 130 and "Column Name Qualifiers in Correlated References" on page 132.

Correlation Names

A *correlation name* can be defined in the FROM clause of a query and in the first clause of an UPDATE or DELETE statement. For example, the clause FROM X.MYTABLE Z establishes Z as a correlation name for X.MYTABLE.

```
  FROM X.MYTABLE Z
```

With Z defined as a correlation name for X.MYTABLE, only Z can be used to qualify a reference to a column of that instance of X.MYTABLE in that SELECT statement.

A correlation name is associated with a table, view, nickname, alias, nested table expression or table function only within the context in which it is defined. Hence, the same correlation name can be defined for different purposes in different statements, or in different clauses of the same statement.

As a qualifier, a correlation name can be used to avoid ambiguity or to establish a correlated reference. It can also be used merely as a shorter name for a table, view, nickname, or alias. In the case of a nested table expression or table function, a correlation name is required to identify the result table. In the example, Z might have been used merely to avoid having to enter X.MYTABLE more than once.

If a correlation name is specified for a table, view, nickname, or alias name, any qualified reference to a column of that instance of the table, view, nickname, or alias must use the correlation name, rather than the table, view, nickname, or alias name. For example, the reference to EMPLOYEE.PROJECT in the following example is incorrect, because a correlation name has been specified for EMPLOYEE:

Example

```
 FROM EMPLOYEE E
   WHERE EMPLOYEE.PROJECT='ABC'      * incorrect*
```

The qualified reference to PROJECT should instead use the correlation name, ″E″, as shown below:

```
 FROM EMPLOYEE E
   WHERE E.PROJECT='ABC'
```

Names specified in a FROM clause are either *exposed* or *non-exposed*. A table, view, nickname, or alias name is said to be exposed in the FROM clause if a correlation name is not specified. A correlation name is always an exposed name. For example, in the following FROM clause, a correlation name is specified for EMPLOYEE but not for DEPARTMENT, so DEPARTMENT is an exposed name, and EMPLOYEE is not:

```
 FROM EMPLOYEE E, DEPARTMENT
```

A table, view, nickname, or alias name that is exposed in a FROM clause may be the same as any other table name, view name or nickname exposed in that

FROM clause or any correlation name in the FROM clause. This may result in ambiguous column name references which returns an error (SQLSTATE 42702).

The first two FROM clauses shown below are correct, because each one contains no more than one reference to EMPLOYEE that is exposed:

1. Given the FROM clause:
   ```
   FROM EMPLOYEE E1, EMPLOYEE
   ```
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the second instance of EMPLOYEE in the FROM clause. A qualified reference to the first instance of EMPLOYEE must use the correlation name "E1" (E1.PROJECT).
2. Given the FROM clause:
   ```
   FROM EMPLOYEE, EMPLOYEE E2
   ```
   a qualified reference such as EMPLOYEE.PROJECT denotes a column of the first instance of EMPLOYEE in the FROM clause. A qualified reference to the second instance of EMPLOYEE must use the correlation name "E2" (E2.PROJECT).
3. Given the FROM clause:
   ```
     FROM EMPLOYEE, EMPLOYEE
   ```
   the two exposed table names included in this clause (EMPLOYEE and EMPLOYEE) are the same. This is allowed, but references to specific column names would be ambiguous (SQLSTATE 42702).
4. Given the following statement:
   ```
   SELECT *
     FROM EMPLOYEE E1, EMPLOYEE E2            * incorrect *
     WHERE EMPLOYEE.PROJECT = 'ABC'
   ```
   the qualified reference EMPLOYEE.PROJECT is incorrect, because both instances of EMPLOYEE in the FROM clause have correlation names. Instead, references to PROJECT must be qualified with either correlation name (E1.PROJECT or E2.PROJECT).
5. Given the FROM clause:
   ```
   FROM EMPLOYEE, X.EMPLOYEE
   ```
   a reference to a column in the second instance of EMPLOYEE must use X.EMPLOYEE (X.EMPLOYEE.PROJECT). If X is the CURRENT SCHEMA special register value in dynamic SQL or the QUALIFIER precompile/bind option in static SQL, then the columns cannot be referenced since any such reference would be ambiguous.

The use of a correlation name in the FROM clause also allows the option of specifying a list of column names to be associated with the columns of the result table. As with a correlation name, these listed column names become the *exposed* names of the columns that must be used for references to the columns throughout the query. If a column name list is specified, then the column names of the underlying table become *non-exposed*.

Given the FROM clause:

```
FROM DEPARTMENT D (NUM,NAME,MGR,ANUM,LOC)
```

a qualified reference such as D.NUM denotes the first column of the DEPARTMENT table that is defined in the table as DEPTNO. A reference to D.DEPTNO using this FROM clause is incorrect since the column name DEPTNO is a non-exposed column name.

Column Name Qualifiers to Avoid Ambiguity

In the context of a function, a GROUP BY clause, ORDER BY clause, an expression, or a search condition, a column name refers to values of a column in some table, view, nickname, nested table expression or table function. The tables, views, nicknames, nested table expressions and table functions that might contain the column are called the *object tables* of the context. Two or more object tables might contain columns with the same name; one reason for qualifying a column name is to designate the table from which the column comes. Qualifiers for column names are also useful in SQL procedures to distinguish column names from SQL variable names used in SQL statements.

A nested table expression or table function will consider *table-references* that precede it in the FROM clause as object tables. The *table-references* that follow are not considered as object tables.

Table Designators

A qualifier that designates a specific object table is called a *table designator*. The clause that identifies the object tables also establishes the table designators for them. For example, the object tables of an expression in a SELECT clause are named in the FROM clause that follows it:

```
SELECT CORZ.COLA, OWNY.MYTABLE.COLA
  FROM OWNX.MYTABLE CORZ, OWNY.MYTABLE
```

Table designators in the FROM clause are established as follows:

- A name that follows a table, view, nickname, alias, nested table expression or table function is both a correlation name and a table designator. Thus, CORZ is a table designator. CORZ is used to qualify the first column name in the select list.
- An exposed table, view name, nickname or alias is a table designator. Thus, OWNY.MYTABLE is a table designator. OWNY.MYTABLE is used to qualify the second column name in the select list.

Each table designator should be unique within a particular FROM clause to avoid the possibility of ambiguous references to columns.

Avoiding Undefined or Ambiguous References

When a column name refers to values of a column, exactly one object table must include a column with that name. The following situations are considered errors:

- No object table contains a column with the specified name. The reference is undefined.
- The column name is qualified by a table designator, but the table designated does not include a column with the specified name. Again the reference is undefined.
- The name is unqualified, and more than one object table includes a column with that name. The reference is ambiguous.
- The column name is qualified by a table designator, but the table designated is not unique in the FROM clause and both occurrences of the designated table include the column. The reference is ambiguous.
- The column name is in a nested table expression which is not preceded by the TABLE keyword or in a table function or nested table expression that is the right operand of a right outer join or a full outer join and the column name does not refer to a column of a *table-reference* within the nested table expression's fullselect. The reference is undefined.

Avoid ambiguous references by qualifying a column name with a uniquely defined table designator. If the column is contained in several object tables with different names, the table names can be used as designators. Ambiguous references can also be avoided without the use of the table designator by giving unique names to the columns of one of the object tables using the column name list following the correlation name.

When qualifying a column with the exposed table name form of a table designator, either the qualified or unqualified form of the exposed table name may be used. However, the qualifier used and the table used must be the same after fully qualifying the table name, view name or nickname and the table designator.

1. If the authorization ID of the statement is CORPDATA:

   ```
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
     FROM EMPLOYEE
   ```

   is a valid statement.
2. If the authorization ID of the statement is REGION:

   ```
   SELECT CORPDATA.EMPLOYEE.WORKDEPT
     FROM EMPLOYEE                         * incorrect *
   ```

is invalid, because EMPLOYEE represents the table REGION.EMPLOYEE, but the qualifier for WORKDEPT represents a different table, CORPDATA.EMPLOYEE.

Column Name Qualifiers in Correlated References

A *fullselect* is a form of a query that may be used as a component of various SQL statements. See "Chapter 5. Queries" on page 393 for more information on fullselects. A fullselect used within a search condition of any statement is called a *subquery*. A fullselect used to retrieve a single value as an expression within a statement is called a *scalar fullselect* or *scalar subquery*. A fullselect used in the FROM clause of a query is called a *nested table expression*. Subqueries in search conditions, scalar subqueries and nested table expressions are referred to as subqueries through the remainder of this topic.

A subquery may include subqueries of its own, and these may, in turn, include subqueries. Thus an SQL statement may contain a hierarchy of subqueries. Those elements of the hierarchy that contain subqueries are said to be at a higher level than the subqueries they contain.

Every element of the hierarchy contains one or more table designators. A subquery can reference not only the columns of the tables identified at its own level in the hierarchy, but also the columns of the tables identified previously in the hierarchy, back to the highest level of the hierarchy. A reference to a column of a table identified at a higher level is called a *correlated reference*.

For compatibility with existing standards for SQL, both qualified and unqualified column names are allowed as correlated references. However, it is good practice to qualify all column references used in subqueries; otherwise, identical column names may lead to unintended results. For example, if a table in a hierarchy is altered to contain the same column name as the correlated reference and the statement is prepared again, the reference will apply to the altered table.

When a column name in a subquery is qualified, each level of the hierarchy is searched, starting at the same subquery as the qualified column name appears and continuing to the higher levels of the hierarchy until a table designator that matches the qualifier is found. Once found, it is verified that the table contains the given column. If the table is found at a higher level than the level containing column name, then it is a correlated reference to the level where the table designator was found. A nested table expression must be preceded with the optional TABLE keyword in order to search the hierarchy above the fullselect of the nested table expression.

When the column name in a subquery is not qualified, the tables referenced at each level of the hierarchy are searched, starting at the same subquery where the column name appears and continuing to higher levels of the hierarchy, until a match for the column name is found. If the column is found in a table at a higher level than the level containing column name, then it is a correlated reference to the level where the table containing the column was found. If the column name is found in more than one table at a particular level, the reference is ambiguous and considered an error.

In either case, T, used in the following example, refers to the table designator that contains column C. A column name, T.C (where T represents either an implicit or an explicit qualifier), is a correlated reference if, and only if, these conditions are met:

- T.C is used in an expression of a subquery.
- T does not designate a table used in the from clause of the subquery.
- T designates a table used at a higher level of the hierarchy that contains the subquery.

Since the same table, view or nickname can be identified at many levels, unique correlation names are recommended as table designators. If T is used to designate a table at more than one level (T is the table name itself or is a duplicate correlation name), T.C refers to the level where T is used that most directly contains the subquery that includes T.C. If a correlation to a higher level is needed, a unique correlation name must be used.

The correlated reference T.C identifies a value of C in a row or group of T to which two search conditions are being applied: condition 1 in the subquery, and condition 2 at some higher level. If condition 2 is used in a WHERE clause, the subquery is evaluated for each row to which condition 2 is applied. If condition 2 is used in a HAVING clause, the subquery is evaluated for each group to which condition 2 is applied. (For another discussion of the evaluation of subqueries, see the descriptions of the WHERE and HAVING clauses in “Chapter 5. Queries” on page 393.)

For example, in the following statement, the correlated reference X.WORKDEPT (in the last line) refers to the value of WORKDEPT in table EMPLOYEE at the level of the first FROM clause. (That clause establishes X as a correlation name for EMPLOYEE.) The statement lists employees who make less than the average salary for their department.

```
SELECT EMPNO, LASTNAME, WORKDEPT
  FROM EMPLOYEE X
  WHERE SALARY < (SELECT AVG(SALARY)
                    FROM EMPLOYEE
                    WHERE WORKDEPT = X.WORKDEPT)
```

The next example uses THIS as a correlation name. The statement deletes rows for departments that have no employees.

```
DELETE FROM DEPARTMENT THIS
   WHERE NOT EXISTS(SELECT *
                      FROM EMPLOYEE
                      WHERE WORKDEPT = THIS.DEPTNO)
```

References to Host Variables

A *host variable* is either:

- A variable in a host language such as a C variable, a C++ variable, a COBOL data item, a FORTRAN variable, or a Java variable

or:

- A host language construct that was generated by an SQL precompiler from a variable declared using SQL extensions

that is referenced in an SQL statement. Host variables are either directly defined by statements in the host language or are indirectly defined using SQL extensions.

A host variable in an SQL statement must identify a host variable described in the program according to the rules for declaring host variables.

All host variables used in an SQL statement must be declared in an SQL DECLARE section in all host languages except REXX (see the *Application Development Guide* for more information on declaring host variables for SQL statements in application programs). No variables may be declared outside an SQL DECLARE section with names identical to variables declared inside an SQL DECLARE section. An SQL DECLARE section begins with BEGIN DECLARE SECTION and ends with END DECLARE SECTION.

The meta-variable *host-variable*, as used in the syntax diagrams, shows a reference to a host variable. A host-variable in the VALUES INTO clause or the INTO clause of a FETCH or a SELECT INTO statement, identifies a host variable to which a value from a column of a row or an expression is assigned. In all other contexts a host-variable specifies a value to be passed to the database manager from the application program.

Host Variables in Dynamic SQL

In dynamic SQL statements, parameter markers are used instead of host variables. A parameter marker is a question mark (?) representing a position in a dynamic SQL statement where the application will provide a value; that is, where a host variable would be found if the statement string were a static SQL statement. The following example shows a static SQL statement using host variables:

```
INSERT INTO DEPARTMENT
   VALUES (:hv_deptno, :hv_deptname, :hv_mgrno, :hv_admrdept)
```

This example shows a dynamic SQL statement using parameter markers:

```
INSERT INTO DEPARTMENT VALUES (?, ?, ?, ?)
```

For more information on parameter markers, see “Parameter Markers” in “PREPARE” on page 954.

The meta-variable *host-variable* in syntax diagrams can generally be expanded to:

```
>>─:host-identifier──┬─────────────────────────────────────┬──────────────><
                     │ ┌─INDICATOR─┐                       │
                     └─┴───────────┴──:host-identifier─────┘
```

Each *host-identifier* must be declared in the source program. The variable designated by the second host-identifier must have a data type of small integer.

The first host-identifier designates the *main variable*. Depending on the operation, it either provides a value to the database manager or is provided a value from the database manager. An input host variable provides a value in the runtime application code page. An output host variable is provided a value that, if necessary, is converted to the runtime application code page when the data is copied to the output application variable. A given host variable can serve as both an input and an output variable in the same program.

The second host-identifier designates its *indicator variable*. The purposes of the indicator variable are to:

- Specify the null value. A negative value of the indicator variable specifies the null value. A value of -2 indicates a numeric conversion or arithmetic expression error occurred in deriving the result
- Record the original length of a truncated string (if the source of the value is not a large object type)
- Record the seconds portion of a time if the time is truncated on assignment to a host variable.

For example, if :HV1:HV2 is used to specify an insert or update value, and if HV2 is negative, the value specified is the null value. If HV2 is not negative the value specified is the value of HV1.

Similarly, if :HV1:HV2 is specified in a VALUES INTO clause or in a FETCH or SELECT INTO statement, and if the value returned is null, HV1 is not changed and HV2 is set to a negative value. [26] If the value returned is not null, that value is assigned to HV1 and HV2 is set to zero (unless the

26. If the database is configured with DFT_SQLMATHWARN yes (or was during binding of a static SQL statement), then HV2 could be -2. If HV2 is -2, then a value for HV1 could not be returned because of an error converting to

assignment to HV1 requires string truncation of a non-LOB string; in which case HV2 is set to the original length of the string). If an assignment requires truncation of the seconds part of a time, HV2 is set to the number of seconds.

If the second host identifier is omitted, the host-variable does not have an indicator variable. The value specified by the host-variable reference :HV1 is always the value of HV1, and null values cannot be assigned to the variable. Thus, this form should not be used in an INTO clause unless the corresponding column cannot contain null values. If this form is used and the column contains nulls, the database manager will generate an error at run time.

An SQL statement that references host variables must be within the scope of the declaration of those host variables. For host variables referenced in the SELECT statement of a cursor, that rule applies to the OPEN statement rather than to the DECLARE CURSOR statement.

Example
Using the PROJECT table, set the host variable PNAME (VARCHAR(26)) to the project name (PROJNAME), the host variable STAFF (dec(5,2)) to the mean staffing level (PRSTAFF), and the host variable MAJPROJ (char(6)) to the major project (MAJPROJ) for project (PROJNO) 'IF1000'. Columns PRSTAFF and MAJPROJ may contain null values, so provide indicator variables STAFF_IND (smallint) and MAJPROJ_IND (smallint).

```
SELECT PROJNAME, PRSTAFF, MAJPROJ
  INTO :PNAME, :STAFF :STAFF_IND, :MAJPROJ :MAJPROJ_IND
  FROM PROJECT
  WHERE PROJNO = 'IF1000'
```

MBCS Considerations: Whether multi-byte characters can be used in a host variable name depends on the host language.

References to BLOB, CLOB, and DBCLOB Host Variables

Regular BLOB, CLOB, and DBCLOB variables, LOB locator variables (see "References to Locator Variables" on page 138), and LOB file reference variables (see "References to BLOB, CLOB, and DBCLOB File Reference Variables" on page 138) can be defined in all host languages. Where LOBs are allowed, the term *host-variable* in a syntax diagram can refer to a regular host variable, a locator variable, or a file reference variable. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

the numeric type of HV1 or an error evaluating an arithmetic expression that is used to determine the value for HV1. When accessing a database with a client version earlier than DB2 Universal Database Version 5, HV2 will be -1 for arithmetic exceptions.

It is sometimes possible to define a large enough variable to hold an entire large object value. If this is true and if there is no performance benefit to be gained by deferred transfer of data from the server, a locator is not needed. However, since host language or space restrictions will often dictate against storing an entire large object in temporary storage at one time and/or because of performance benefit, a large object may be referenced via a locator and portions of that object may be selected into or updated from host variables that contain only a portion of the large object at one time.

As with all other host variables, a large object locator variable may have an associated indicator variable. Indicator variables for large object locator host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the locator host variable is unchanged. This means a locator can never point to a null value.

References to Locator Variables

A *locator variable* is a host variable that contains the locator representing a LOB value on the application server. (See “Manipulating Large Objects (LOBs) with Locators” on page 77 for information on how locators can be used to manipulate LOB values.)

A locator variable in an SQL statement must identify a locator variable described in the program according to the rules for declaring locator variables. This is always indirectly through an SQL statement.

The term locator variable, as used in the syntax diagrams, shows a reference to a locator variable. The meta-variable *locator-variable* can be expanded to include a *host-identifier* the same as that for *host-variable*.

When the indicator variable associated with a locator is null, the value of the referenced LOB is null.

If a locator-variable that does not currently represent any value is referenced, an error is raised (SQLSTATE 0F001).

At transaction commit, or any transaction termination, all locators acquired by that transaction are released.

References to BLOB, CLOB, and DBCLOB File Reference Variables

BLOB, CLOB, and DBCLOB file reference variables are used for direct file input and output for LOBs, and can be defined in all host languages. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable. In the case of REXX, LOBs are mapped to strings.

A file reference variable represents (rather than contains) the file, just as a LOB locator represents, rather than contains, the LOB bytes. Database queries, updates and inserts may use file reference variables to store or to retrieve single column values.

A file reference variable has the following properties:

Data Type
BLOB, CLOB, or DBCLOB. This property is specified when the variable is declared.

Direction
This must be specified by the application program at run time (as part of the File Options value). The direction is one of:

- Input (used as a source of data on an EXECUTE statement, an OPEN statement, an UPDATE statement, an INSERT statement, or a DELETE statement).
- Output (used as the target of data on a FETCH statement or a SELECT INTO statement).

File name
This must be specified by the application program at run time. It is one of:

- The complete path name of the file (which is advised).
- A relative file name. If a relative file name is provided, it is appended to the current path of the client process.

Within an application, a file should only be referenced in one file reference variable.

File Name Length
This must be specified by the application program at run time. It is the length of the file name (in bytes).

File Options
An application must assign one of a number of options to a file reference variable before it makes use of that variable. Options are set by an INTEGER value in a field in the file reference variable structure. One of the following values must be specified for each file reference variable:

- Input (from client to server)

SQL_FILE_READ [27]
This is a regular file that can be opened, read and closed.

- Output (from server to client)

SQL_FILE_CREATE [28]
Create a new file. If the file already exists, it is an error.

SQL_FILE_OVERWRITE (Overwrite) [29]
If an existing file with the specified name exists, it is overwritten; otherwise a new file is created.

SQL_FILE_APPEND [30]
If an existing file with the specified name exists, the output is appended to it; otherwise a new file is created.

Data Length
This is unused on input. On output, the implementation sets the data length to the length of the new data written to the file. The length is in bytes.

As with all other host variables, a file reference variable may have an associated indicator variable.

Example of an Output File Reference Variable (in C)

- Given a declare section is coded as:

```
EXEC SQL BEGIN DECLARE SECTION
   SQL TYPE IS CLOB_FILE  hv_text_file;
   char  hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

  Following preprocessing this would be:

27. SQL-FILE-READ in COBOL, sql_file_read in FORTRAN, READ in REXX.
28. SQL-FILE-CREATE in COBOL, sql_file_create in FORTRAN, CREATE in REXX.
29. SQL-FILE-OVERWRITE in COBOL, sql_file_overwrite in FORTRAN, OVERWRITE in REXX.
30. SQL-FILE-APPEND in COBOL, sql_file_append in FORTRAN, APPEND in REXX.

```
EXEC SQL BEGIN DECLARE SECTION
   /* SQL TYPE IS CLOB_FILE  hv_text_file; */
   struct {
       unsigned long  name_length; //  File Name Length
       unsigned long  data_length; //  Data Length
       unsigned long  file_options; // File Options
       char           name[255];   // File Name
   } hv_text_file;
   char  hv_patent_title[64];
EXEC SQL END DECLARE SECTION
```

Then, the following code can be used to select from a CLOB column in the database into a new file referenced by :hv_text_file.

```
strcpy(hv_text_file.name, "/u/gainer/papers/sigmod.94");
hv_text_file.name_length = strlen("/u/gainer/papers/sigmod.94");
hv_text_file.file_options = SQL_FILE_CREATE;

EXEC SQL SELECT content INTO :hv_text_file from papers
     WHERE TITLE = 'The Relational Theory behind Juggling';
```

Example of an Input File Reference Variable (in C)

- Given the same declare section as above, the following code can be used to insert the data from a regular file referenced by :hv_text_file into a CLOB column.

```
 strcpy(hv_text_file.name, "/u/gainer/patents/chips.13");
hv_text_file.name_length = strlen("/u/gainer/patents/chips.13");
hv_text_file.file_options = SQL_FILE_READ:
strcpy(:hv_patent_title, "A Method for Pipelining Chip Consumption");

EXEC SQL INSERT INTO patents( title, text )
         VALUES(:hv_patent_title, :hv_text_file);
```

References to Structured Type Host Variables

Structured type variables can be defined in all host languages except FORTRAN, REXX, and Java. Since these are not native data types, SQL extensions are used and the precompilers generate the host language constructs necessary to represent each variable.

As with all other host variables, a structured type variable may have an associated indicator variable. Indicator variables for structured type host variables behave in the same way as indicator variables for other data types. When a null value is returned from the database, the indicator variable is set and the structured type host variable is unchanged.

The actual host variable for a structured type is defined as a built-in data type. The built-in data type associated with the structured type must be assignable:

- from the result of the FROM SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command; and
- to the parameter of the TO SQL transform function for the structured type as defined by the specified TRANSFORM GROUP option of the precompile command.

If using a parameter marker instead of a host variable, the appropriate parameter type characteristics must be specified in the SQLDA. This requires a ″doubled″ set of SQLVAR structures in the SQLDA, and the SQLDATATYPE_NAME field of the secondary SQLVAR must be filled with the schema and type name of the structured type. If the schema is omitted in the SQLDA structure, an error results (SQLSTATE 07002). For additional information on this topic, see "Appendix C. SQL Descriptor Area (SQLDA)" on page 1113.

Example

Define the host variables *hv_poly* and *hv_point* (of type POLYGON, using built-in type BLOB(1048576)) in a C program.

```
EXEC SQL BEGIN DECLARE SECTION;
      static SQL
         TYPE IS POLYGON AS BLOB(1M)
         hv_poly, hv_point;
EXEC SQL END DECLARE SECTION;
```

Functions

A *database function* is a relationship between a set of input data values and a set of result values. For example, the TIMESTAMP function can be passed input data values of type DATE and TIME and the result is a TIMESTAMP. Functions can either be built-in or user-defined.

- *Built-in functions* are provided with the database manager providing a single result value and are identified as part of the SYSIBM schema. Examples of such functions include column functions such as AVG, operator functions such as ″+″, casting functions such as DECIMAL, and others such as SUBSTR.
- *User-defined functions* are functions that are registered to a database in SYSCAT.FUNCTIONS (using the CREATE FUNCTION statement). User-defined functions are never part of the SYSIBM schema. One such set of functions is provided with the database manager in a schema called SYSFUN.

With user-defined functions, DB2 allows users and application developers to extend the function of the database system by adding function definitions provided by users or third party vendors to be applied in the database engine itself. This allows higher performance than retrieving rows from the database and applying those functions on the retrieved data to further qualify or to perform data reduction. Extending database functions also lets the database exploit the same functions in the engine that an application uses, provides more synergy between application and database, and contributes to higher productivity for application developers because it is more object-oriented.

A complete list of functions in the SYSIBM and SYSFUN schemas is documented in Table 15 on page 210.

External, SQL and Sourced User-Defined Functions

A user-defined function can be an *external function*, an *SQL function*, or a *sourced function*. An *external function* is defined to the database with a reference to an object code library and a function within that library that will be executed when the function is invoked. External functions can not be column functions. A *sourced function* is defined to the database with a reference to another built-in or user-defined function that is already known to the database. Sourced functions can be scalar functions or column functions. They are very useful for supporting the use of existing functions with user-defined types. An *SQL function* is defined to the database using only the SQL RETURN statement. It can return either a scalar value, a row, or a table. SQL functions cannot be column functions.

Scalar, Column, Row and Table User-Defined Functions

Each user-defined function is also categorized as a *scalar*, *column* or *table* function.

A *scalar function* is one which returns a single-valued answer each time it is called. For example, the built-in function SUBSTR() is a scalar function. Scalar UDFs can be either external or sourced.

A *column function* is one which conceptually is passed a set of like values (a column) and returns a single-valued answer. These are also sometimes called *aggregating functions* in DB2. An example of a column function is the built-in function AVG(). An external column UDF cannot be defined to DB2, but a column UDF which is sourced upon one of the built-in column functions can be defined. This is useful for distinct types. For example if there is a distinct type SHOESIZE defined with base type INTEGER, a UDF AVG(SHOESIZE) which is sourced on the built-in function AVG(INTEGER) could be defined, and it would be a column function.

A *row function* is a function which returns one row of values. It may only be used as a transform function, mapping attribute values of a structured type into values in a row. A row function must be defined as an SQL function.

A *table function* is a function which returns a table to the SQL statement which references it. It may only be referenced in the FROM clause of a SELECT. Such a function can be used to apply SQL language processing power to data which is not DB2 data, or to convert such data into a DB2 table. It could, for example, take a file and convert it to a table, sample data from the World Wide Web and tabularize it, or access a Lotus Notes database and return information about mail messages, such as the date, sender, and the text of the message. This information can be joined with other tables in the database. A table function can be defined as an external function or as an SQL function (a table function cannot be a sourced function).

Function signatures

A function is identified by its schema, a function name, the number of parameters and the data types of its parameters. This is called a *function signature* which must be unique within the database. There can be more than one function with the same name in a schema provided that the number of parameters or the data types of the parameters are different. A function name for which there are multiple function instances is called an *overloaded* function. A function name can be overloaded within a schema, in which case there is more than one function by that name in the schema (which of necessity have different parameter types). A function name can also be overloaded in a SQL path, in which case there is more than one function by that name in the path, and these functions do not necessarily have different parameter types.

SQL Path

A function can be invoked by referring, in an allowable context, to the qualified name (schema and function name) followed by the list of arguments enclosed in parentheses. A function can also be invoked without the schema name resulting in a choice of possible functions in different schemas with the same or acceptable parameters. In this case, the *SQL path* is used to assist in *function resolution*. The SQL path is a list of schemas that are searched to identify a function with the same name, number of parameters and acceptable data types. For static SQL statements, SQL path is specified using the FUNCPATH bind option (see *Command Reference* for details). For dynamic SQL statements, SQL path is the value of the CURRENT PATH special register (see "CURRENT PATH" on page 122).

Function Resolution

Given a function invocation, the database manager must decide which of the possible functions with the same name is the "best" fit. This includes resolving functions from the built-in and user-defined functions.

An *argument* is a value passed to a function upon invocation. When a function is invoked in SQL, it is passed a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. A *parameter* is a formal definition of an input to a function. When a function is defined to the database, either internally (the built-in functions) or by a user (user-defined functions), its parameters (zero or more) are specified, the order of their definitions defining their positions and thus their semantics. Therefore, every parameter is a particular positional input of a function. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the function given in the invocation, the number and data types of the arguments, all the functions with the same name in the SQL path, and the data types of their corresponding parameters as the basis for deciding whether or not to select a function. The following are the possible outcomes of the decision process:

1. A particular function is deemed to be the best fit. For example, given the functions named RISK in the schema TEST with signatures defined as:

   ```
   TEST.RISK(INTEGER)
   TEST.RISK(DOUBLE)
   ```

   a SQL path including the TEST schema and the following function reference (where DB is a DOUBLE column):

   ```
   SELECT ... RISK(DB) ...
   ```

   then, the second RISK will be chosen.

   The following function reference (where SI is a SMALLINT column):

   ```
   SELECT ... RISK(SI) ...
   ```

   would choose the first RISK, since SMALLINT can be promoted to INTEGER and is a better match than DOUBLE which is further down the precedence list (as shown in Table 5 on page 90).

   When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter closest in the structured type hierarchy to the static type of the function argument.
2. No function is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

   ```
   SELECT ... RISK(C) ...
   ```

   the argument is inconsistent with the parameter of both RISK functions.
3. A particular function is selected based on the SQL path and the number and data types of the arguments passed on invocation. For example, given functions named RANDOM with signatures defined as:

   ```
   TEST.RANDOM(INTEGER)
   PROD.RANDOM(INTEGER)
   ```

   and a SQL path of:

   ```
   "TEST","PROD"
   ```

Then the following function reference:

```
   SELECT ... RANDOM(432) ...
```

will choose TEST.RANDOM since both RANDOM functions are equally good matches (exact matches in this particular case) and both schemas are in the path, but TEST precedes PROD in the SQL path.

Method of Choosing the Best Fit

A comparison of the data types of the arguments with the defined data types of the parameters of the functions under consideration forms the basis for the decision of which function in a group of like-named functions is the ″best fit″. Note that the data type of the result of the function or the type of function (column, scalar, or table) under consideration **does not** enter into this determination.

Function resolution is done using the steps that follow.

1. First, find all functions from the catalog (SYSCAT.FUNCTIONS) and built-in functions such that all of the following are true:
   a. For invocations where the schema name was specified (i.e. qualified references), then the schema name and the function name match the invocation name.
   b. For invocations where the schema name was not specified (i.e. unqualified references), then the function name matches the invocation name and has a schema name that matches one of the schemas in the SQL path.
   c. The number of defined parameters matches the invocation.
   d. Each invocation argument matches the function’s corresponding defined parameter in data type, or is “promotable” to it (see “Promotion of Data Types” on page 90).
2. Next, consider each argument of the function invocation, from left to right. For each argument, eliminate all functions that are not the *best match* for that argument. The *best match* for a given argument is the first data type appearing in the precedence list corresponding to the argument data type in Table 5 on page 90 for which there exists a function with a parameter of that data type. Lengths, precisions, scales and the ″FOR BIT DATA″ attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

   The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Note that only the static type (declared type) of the structured-type argument is considered, not the dynamic type (most specific type).

3. If more than one candidate function remains after Step 2, then it has to be the case (the way the algorithm works) that all the remaining candidate functions have identical signatures but are in different schemas. Choose the function whose schema is earliest in the user's SQL path.
4. If there are no candidate functions remaining after step 2, an error is returned (SQLSTATE 42884).

Function Path Considerations for Built-in Functions

Built-in functions reside in a special schema called SYSIBM. Additional functions are available in the SYSFUN schema which are not considered built-in functions since they are developed as user-defined functions and have no special processing considerations. Users can not define additional functions in SYSIBM or SYSFUN schemas (or in any schema whose name begins with the letters "SYS").

As already stated, the built-in functions participate in the function resolution process exactly as do the user-defined functions. One difference between built-in and user-defined functions, from a function resolution perspective, is that the built-in function must always be considered by function resolution. Therefore, omission of SYSIBM from the path results in an assumption for function and data type resolution that SYSIBM is the first schema on the path.

For example, if a user's SQL path is defined as:

```
"SHAREFUN","SYSIBM","SYSFUN"
```

and there is a LENGTH function defined in schema SHAREFUN with the same number and types of arguments as SYSIBM.LENGTH, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SHAREFUN.LENGTH. However, if the user's SQL path is defined as:

```
"SHAREFUN","SYSFUN"
```

and the same SHAREFUN.LENGTH function exists, then an unqualified reference to LENGTH in this user's SQL statement will result in selecting SYSIBM.LENGTH since SYSIBM is implicitly first in the path because it was not specified.

It is possible to minimize potential problems in this area by:
- never using the names of built-in functions for user-defined functions, or
- qualifying any references to these functions, if for some reason it is deemed necessary to create a user-defined function with the same name as a built-in function.

Example of Function Resolution

The following is an example of successful function resolution.

There are seven FOO functions, in three different schemas, registered as (note that not all required keywords appear):

```
CREATE FUNCTION AUGUSTUS.FOO (CHAR(5), INT, DOUBLE)  SPECIFIC FOO_1 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE)      SPECIFIC FOO_2 ...
CREATE FUNCTION AUGUSTUS.FOO (INT, INT, DOUBLE, INT) SPECIFIC FOO_3 ...
CREATE FUNCTION JULIUS.FOO   (INT, DOUBLE, DOUBLE)   SPECIFIC FOO_4 ...
CREATE FUNCTION JULIUS.FOO   (INT, INT, DOUBLE)      SPECIFIC FOO_5 ...
CREATE FUNCTION JULIUS.FOO   (SMALLINT, INT, DOUBLE) SPECIFIC FOO_6 ...
CREATE FUNCTION NERO.FOO     (INT, INT, DEC(7,2))    SPECIFIC FOO_7 ...
```

The function reference is as follows (where I1 and I2 are INTEGER columns, and D is a DECIMAL column):

```
    SELECT ... FOO(I1, I2, D) ...
```

Assume that the application making this reference has a SQL path established as:

```
    "JULIUS","AUGUSTUS","CAESAR"
```

Following through the algorithm...

FOO_7 is eliminated as a candidate, because the schema ″NERO″ is not included in the SQL path.

FOO_3 is eliminated as a candidate, because it has the wrong number of parameters. FOO_1 and FOO_6 are eliminated because in both cases the first argument cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are then considered in order.

For the first argument, all remaining functions — FOO_2, FOO_4 and FOO_5 are an exact match with the argument type. No functions can be eliminated from consideration, therefore the next argument must be examined.

For this second argument, FOO_2 and FOO_5 are exact matches while FOO_4 is not, so it is eliminated from consideration. The next argument is examined to determine some differentiation between FOO_2 and FOO_5.

On the third and last argument, neither FOO_2 nor FOO_5 match the argument type exactly, but both are equally good.

There are two functions remaining, FOO_2 and FOO_5, with identical parameter signatures. The final tie-breaker is to see which function's schema comes first in the SQL path, and on this basis FOO_5 is finally chosen.

Function Invocation

Once the function is selected, there are still possible reasons why the use of the function may not be permitted. Each function is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the function is invoked, an error will occur. For example, given functions named STEP defined, this time, with different data types as the result:

```
STEP(SMALLINT) returns CHAR(5)
STEP(DOUBLE)   returns INTEGER
```

and the following function reference (where S is a SMALLINT column):

```
SELECT ... 3 + STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

A couple of other examples where this can happen are as follows, both of which will result in an error on the statement:

1. The function was referenced in a FROM clause, but the function selected by the function resolution step was a scalar or column function.
2. The reverse case, where the context calls for a scalar or column function, and function resolution selects a table function.

In cases where the arguments of the function invocation were not an exact match to the data types of the parameters of the selected function, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns (see "Assignments and Comparisons" on page 94). This includes the case where precision, scale, or length differs between the argument and the parameter.

Methods

A database method of a structured type is a relationship between a set of input data values and a set of result values, where the first input value (or *subject argument*) has the same type, or is a subtype of the subject type (also called the *subject parameter*), of the method. For example, a method called CITY, of type ADDRESS, can be passed input data values of type VARCHAR and the result is an ADDRESS (or a subtype of ADDRESS).

Methods are defined implicitly or explicitly, as part of the definition of a user-defined structured type.

Implicitly defined methods are created for every structured type. Observer methods are defined for each attribute of the structured type. Observer methods allow applications to get the value of an attribute for an instance of the type. Mutator methods are also defined for each attribute, allowing applications to mutate the type instance by changing the value for an attribute of a type instance. The CITY method described above is an example of a mutator method for the type ADDRESS.

Methods

Explicitly defined methods, or *user-defined methods* are methods that are registered to a database in SYSCAT.FUNCTIONS, by using a combination of CREATE TYPE (or ALTER TYPE ADD METHOD) and CREATE METHOD statements. All methods defined for a structured type are defined in the same schema as the type.

With user-defined methods for structured types, DB2 allows users and application developers to extend the function of the database system. This is accomplished by adding method definitions provided by users, or third party vendors, to be applied to structured type instances in the database engine. The added method definitions provide a higher level of performance as opposed to retrieving rows from the database and applying functions on the retrieved data. Defining database methods also enables the database to exploit the same methods in the engine used by an application, providing a greater degree of interaction efficiency between the application and database. This results in higher productivity for application developers, because it is more object-oriented.

External and SQL User-Defined Methods

A user-defined method can be either external or based on an SQL expression. An external method is defined to the database with a reference to an object code library and a function within that library that will be executed when the method is invoked. A method based on an SQL expression returns the result of the SQL expression when the method is invoked. Such methods do not require any object code library, since they are written completely in SQL.

A user-defined method can return a single-valued answer each time it is called. This value can be a structured type. A method can be defined as *type preserving* (using SELF AS RESULT), to allow the dynamic type of the subject argument to be returned as the returned type of the method. All implicitly defined mutator methods are type preserving.

Method Signatures

A method is identified by its subject type, a method name, the number of parameters and the data types of its parameters. This is called a method signature, and it must be unique within the database.

There can be more than one method with the same name for a structured type provided that:

- the number of parameters or the data types of the parameters are different
- the same signature does not exist for a subtype or supertype of the subject type of the method
- the same function signature (using the subject type or any of its subtypes or supertypes as the first parameter) does not exist.

A method name which has multiple method instances is called an *overloaded method*. A method name can be overloaded within a type, in which case there is more than one method by that name for the type (all of which have different parameter types). A method name can also be overloaded in the subject type hierarchy, in which case there is more than one method by that name in the type hierarchy, and these methods also must have different parameter types.

Method Invocation

A method can be invoked by referring, in an allowable context, to the method name preceded by both a reference to a structured type instance (the subject argument), and the double dot operator. The list of arguments enclosed in parentheses must follow. The method that is actually invoked is determined based on the static type of the subject type, using the method resolution described in the following section. Methods defined WITH FUNCTION ACCESS can also be invoked using function invocation, in which case the regular rules for function resolution are applied.

Method Resolution

Given a method invocation, the database manager must decide which of the possible methods with the same name is the ″best″ fit. Functions (built-in or user-defined) are not considered during method resolution.

An argument is a value passed to a method upon invocation. When a method is invoked in SQL, it is passed the subject argument (of some structured type) and a list of zero or more arguments. They are positional in that the semantics of an argument are determined by its position in the argument list. The subject argument is considered as the first argument. A parameter is a formal definition of an input to a method.

When a method is defined to the database, either implicitly (system-generated for a type) or by a user (user-defined methods), its parameters are specified (with the subject parameter as the first parameter), and the order of their definitions defines their positions and their semantics. Therefore, every parameter is a particular positional input of a method. On invocation, an argument corresponds to a particular parameter by virtue of its position in the list of arguments.

The database manager uses the name of the method given in the invocation, the number and data types of the arguments, all the methods with the same name for the subject argument's static type (and it's supertypes), and the data types of their corresponding parameters as the basis for deciding whether or not to select a method.

The following are the possible outcomes of the decision process:

1. A particular method is deemed to be the best fit. For example, given the methods named RISK for the type SITE with signatures defined as:

   ```
   PROXIMITY(INTEGER) FOR SITE
   PROXIMITY(DOUBLE) FOR SITE
   ```

   the following method invocation (where ST is a SITE column, DB is a DOUBLE column):

   ```
   SELECT ST..PROXIMITY(DB) ...
   ```

   then, the second PROXIMITY will be chosen.

   The following method invocation (where SI is a SMALLINT column):

   ```
   SELECT ST..PROXIMITY(SI) ...
   ```

   would choose the first PROXIMITY, since SMALLINT can be promoted to INTEGER and is a better match than DOUBLE, which is further down the precedence list.

   When considering arguments that are structured types, the precedence list includes the supertypes of the static type of the argument. The best fit is the function defined with the supertype parameter that is closest in the structured type hierarchy to the static type of the function argument.
2. No method is deemed to be an acceptable fit. For example, given the same two functions in the previous case and the following function reference (where C is a CHAR(5) column):

   ```
   SELECT ST..PROXIMITY(C) ...
   ```

   the argument is inconsistent with the parameter of both PROXIMITY functions.
3. A particular method is selected based on the methods in the type hierarchy and the number and data types of the arguments passed on invocation. For example, given the methods named RISK for the types SITE and DRILLSITE (a subtype of SITE) with signatures defined as:

   ```
   RISK(INTEGER) FOR DRILLSITE
   RISK(DOUBLE) FOR SITE
   ```

   the following method invocation (where DRST is a DRILLSITE column, DB is a DOUBLE column):

   ```
   SELECT DRST..RISK(DB) ...
   ```

   then, the second RISK will be chosen since DRILLSITE can be promoted to SITE.

   The following method reference (where SI is a SMALLINT column):

   ```
   SELECT DRST..RISK(SI) ...
   ```

would choose the first RISK, since SMALLINT can be promoted to INTEGER, which is closer on the precedence list than DOUBLE, and DRILLSITE is a better match than SITE, which is a supertype.

Methods within the same type hierarchy cannot have the same signatures, considering the parameters other than the subject parameter.

Method of Choosing the Best Fit

Comparing the data types of the arguments with the defined data types of the parameters of the method under consideration forms the basis for the decision of which method in a group of like-named methods is the ″best fit″. Note that the data type of the result of the method under consideration does not enter into this determination.

Method resolution is done using the steps that follow.

1. First, find all methods from the catalog (SYSCAT.FUNCTIONS) such that all of the following are true:
   - the method name matches the invocation name, and the subject parameter is the same type or is a supertype of the static type of the subject argument
   - the number of defined parameters matches the invocation
   - each invocation argument matches the method’s corresponding defined parameter in data type, or is ″promotable″ to it (see “Promotion of Data Types” on page 90).
2. Next, consider each argument of the method invocation, from left to right. The leftmost argument (and thus the first argument) is the implicit SELF parameter. For example, a method defined for type ADDRESS_T has an implicit first parameter of type ADDRESS_T.

   For each argument, eliminate all functions that are not the best match for that argument. The best match for a given argument is the first data type appearing in the precedence list, corresponding to the argument data type in Table 5 on page 90 for which there exists a function with a parameter of that data type.

   Lengths, precisions, scales and the ″FOR BIT DATA″ attribute are not considered in this comparison. For example, a DECIMAL(9,1) argument is considered an exact match for a DECIMAL(6,5) parameter, and a VARCHAR(19) argument is an exact match for a VARCHAR(6) parameter.

   The best match for a user-defined structured-type argument is itself; the next best match is its immediate supertype, and so on for each supertype of the argument. Notice that only the static type (declared type) of the structured-type argument is considered, and not the dynamic type (most specific type).
3. At most, one candidate method remains after Step 2. This is the method that is chosen.

4. If there are no candidate methods remaining after step 2, an error is returned (SQLSTATE 42884).

Example of Method Resolution

The following is an example of successful method resolution.

There are seven FOO methods for three structured types defined in a hierarchy of GOVERNOR as a subtype of EMPEROR as a subtype of HEADOFSTATE, registered with signatures:

```
CREATE METHOD FOO (CHAR(5), INT, DOUBLE)  FOR HEADOFSTATE SPECIFIC FOO_1 ...
CREATE METHOD FOO (INT, INT, DOUBLE)      FOR HEADOFSTATE SPECIFIC FOO_2 ...
CREATE METHOD FOO (INT, INT, DOUBLE, INT) FOR HEADOFSTATE SPECIFIC FOO_3 ...
CREATE METHOD FOO (INT, DOUBLE, DOUBLE)   FOR EMPEROR     SPECIFIC FOO_4 ...
CREATE METHOD FOO (INT, INT, DOUBLE)      FOR EMPEROR     SPECIFIC FOO_5 ...
CREATE METHOD FOO (SMALLINT, INT, DOUBLE) FOR EMPEROR     SPECIFIC FOO_6 ...
CREATE METHOD FOO (INT, INT, DEC(7,2))    FOR GOVERNOR    SPECIFIC FOO_7 ...
```

The method reference is as follows (where I1 and I2 are INTEGER columns, D is a DECIMAL column and E is an EMPEROR column):

```
SELECT E..FOO(I1, I2, D) ...
```

Following through the algorithm...

FOO_7 is eliminated as a candidate, because the type GOVERNOR is a subtype of EMPEROR (not a supertype).

FOO_3 is eliminated as a candidate, because it has the wrong number of parameters.

FOO_1 and FOO_6 are eliminated because, in both cases, the first argument (not the subject argument) cannot be promoted to the data type of the first parameter. Because there is more than one candidate remaining, the arguments are then considered in order.

For the subject argument, FOO_2 is a supertype, while FOO_4 and FOO_5 match the subject argument.

For the first argument, the remaining methods, FOO_4 and FOO_5, are an exact match with the argument type. No methods can be eliminated from consideration, therefore the next argument must be examined.

For this second argument, FOO_5 is an exact match while FOO_4 is not, so it is eliminated from consideration. This leaves FOO_5 as the method chosen.

Method Invocation

Once the method is selected, there are still possible reasons why the use of the method may not be permitted.

Each method is defined to return a result with a specific data type. If this result data type is not compatible with the context in which the method is invoked, an error will occur. For example, assume that the following methods named STEP are defined, each with a different data type as the result:

```
STEP(SMALLINT) FOR TYPEA RETURNS CHAR(5)
STEP(DOUBLE) FOR TYPEA RETURNS INTEGER
```

and the following method reference (where S is a SMALLINT column and TA is an column of TYPEA):

```
SELECT 3 + TA..STEP(S) ...
```

then, because there is an exact match on argument type, the first STEP is chosen. An error occurs on the statement, because the result type is CHAR(5) instead of a numeric type as required for an argument of the addition operator.

Note that when the selected method is a type preserving method:

- the static result type following function resolution is the same as the static type of the subject argument of the method invocation
- the dynamic result type when the method is invoked is the same as the dynamic type of the subject argument of the method invocation.

This may be a subtype of the result type specified in the type preserving method definition, which in turn may be a supertype of the dynamic type that is actually returned when the method is processed.

In cases where the arguments of the method invocation were not an exact match to the data types of the parameters of the selected method, the arguments are converted to the data type of the parameter at execution using the same rules as assignment to columns (see "Assignments and Comparisons" on page 94). This includes the case where precision, scale, or length differs between the argument and the parameter, but excludes the case where the dynamic type of the argument is a subtype of the parameter's static type.

Conservative Binding Semantics

There are situations within a database where the functions, methods and data types are resolved when the statement is processed and the database manager must be able to repeat this resolution. This is true in:

- static DML statements in packages,
- views,
- triggers,
- check constraints, and
- SQL routines.

Conservative Binding Semantics

For static DML statements in packages, the function, method and data type references are resolved during the bind operation. Function, method and data type references in views, triggers, and check constraints are resolved when the database object is created.

If function or method resolution is performed again on any function or method references in these objects, it could change the behavior if a new function or method has been added with a signature that is a better match but the actual executable performs different operations. Similarly, if resolution is performed again on any data type in these objects, it could change the behavior if a new data type has been added with the same name in a different schema that is also on the SQL path. In order to avoid this, where necessary, the database manager applies the concept of *conservative binding semantics*. This concept ensures that the function and data type references will be resolved using the same SQL path as when it was bound. Furthermore, the creation timestamp of functions [31], methods and data types considered during resolution is not later than the time when the statement was bound [32]. In this way, only the functions and data types that were considered during function, method and data type resolution when the statement was originally processed will be considered. Hence, newly created functions, methods and data types are not considered when conservative binding semantics are applied.

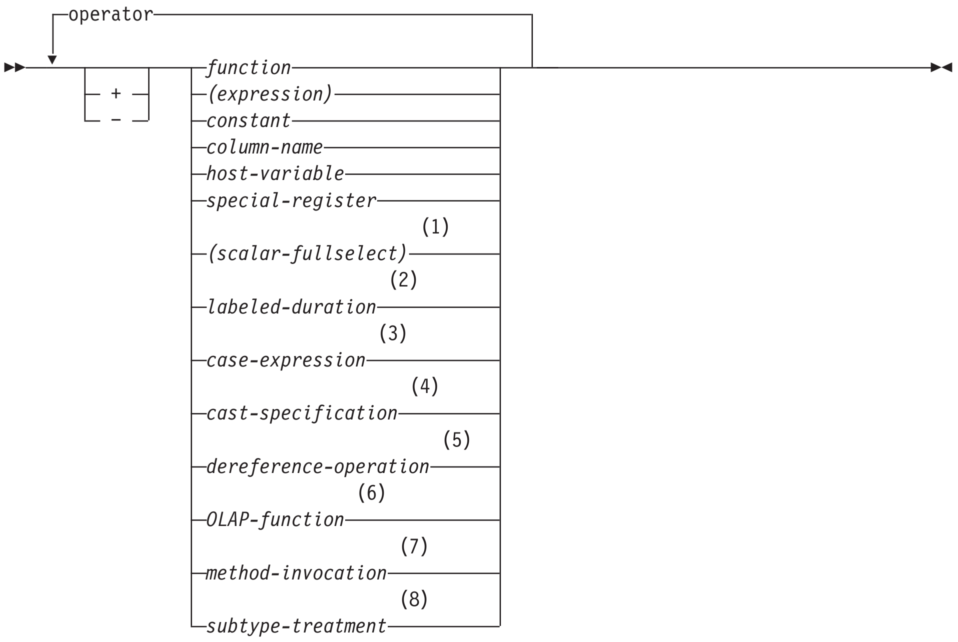
For static DML in packages, the packages can rebind either implicitly or by explicitly issuing the REBIND command (or corresponding API) or the BIND command (or corresponding API). The implicit rebind is always performed to resolve functions, methods and data types with conservative binding semantics. The REBIND command provides the choice to resolve with conservative binding semantics (RESOLVE CONSERVATIVE option) or to resolve considering any new functions, methods and data types (by default or using the RESOLVE ANY option).

31. Built-in functions added starting with Version 6.1 have a creation timestamp that is based on the time of database creation or migration.

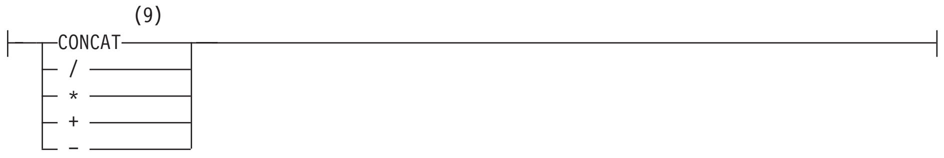
32. For views, conservative binding also ensures that the columns of the view are the same as those that existed at the time the view was created. For example, a view defined using the asterisk in the select list will not consider any columns added to the underlying tables after the view was created.

Expressions

An expression specifies a value. It can be a simple value, consisting of only a constant or a column name, or it can be more complex. When repeatedly using similar complex expressions, the usage of an SQL function may be considered to encapsulate a common expression. See “CREATE FUNCTION (SQL Scalar, Table or Row)” on page 649 for more information.



operator:



Notes:

1 See “Scalar Fullselect” on page 164 for more information.

2 See “Labeled Durations” on page 164 for more information.

3 See “CASE Expressions” on page 171 for more information.

4 See “CAST Specifications” on page 173 for more information.

5 See “Dereference Operations” on page 176 for more information.

6 See “OLAP Functions” on page 177 for more information.

7 See “Method Invocation” on page 183 for more information.

8 See “Subtype Treatment” on page 184 for more information.

9 `||` may be used as a synonym for CONCAT.

Without Operators

If no operators are used, the result of the expression is the specified value.

Examples: `SALARY` `:SALARY` `'SALARY'` `**MAX**(SALARY)`

With the Concatenation Operator

The concatenation operator (CONCAT) links two string operands to form a *string expression*.

The operands of concatenation must be compatible strings. Note that a binary string cannot be concatenated with a character string, including character strings defined as FOR BIT DATA (SQLSTATE 42884). For more information on compatibility, refer to the compatibility matrix on page Table 7 on page 94.

If either operand can be null, the result can be null, and if either is null, the result is the null value. Otherwise, the result consists of the first operand string followed by the second. Note that no check is made for improperly formed mixed data when doing concatenation.

The length of the result is the sum of the lengths of the operands.

The data type and length attribute of the result is determined from that of the operands as shown in the following table:

Table 10. Data Type and Length of Concatenated Operands

Operands	Combined Length Attributes	Result
CHAR(A) CHAR(B)	<255	CHAR(A+B)
CHAR(A) CHAR(B)	>254	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
CHAR(A) VARCHAR(B)	>4000	LONG VARCHAR
CHAR(A) LONG VARCHAR	-	LONG VARCHAR
VARCHAR(A) VARCHAR(B)	<4001	VARCHAR(A+B)
VARCHAR(A) VARCHAR(B)	>4000	LONG VARCHAR

Table 10. Data Type and Length of Concatenated Operands (continued)

Operands	Combined Length Attributes	Result
VARCHAR(A) LONG VARCHAR	-	LONG VARCHAR
LONG VARCHAR LONG VARCHAR	-	LONG VARCHAR
CLOB(A) CHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) VARCHAR(B)	-	CLOB(MIN(A+B, 2G))
CLOB(A) LONG VARCHAR	-	CLOB(MIN(A+32K, 2G))
CLOB(A) CLOB(B)	-	CLOB(MIN(A+B, 2G))
GRAPHIC(A) GRAPHIC(B)	<128	GRAPHIC(A+B)
GRAPHIC(A) GRAPHIC(B)	>127	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
GRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
GRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
VARGRAPHIC(A) VARGRAPHIC(B)	<2001	VARGRAPHIC(A+B)
VARGRAPHIC(A) VARGRAPHIC(B)	>2000	LONG VARGRAPHIC
VARGRAPHIC(A) LONG VARGRAPHIC	-	LONG VARGRAPHIC
LONG VARGRAPHIC LONG VARGRAPHIC	-	LONG VARGRAPHIC
DBCLOB(A) GRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) VARGRAPHIC(B)	-	DBCLOB(MIN(A+B, 1G))
DBCLOB(A) LONG VARGRAPHIC	-	DBCLOB(MIN(A+16K, 1G))
DBCLOB(A) DBCLOB(B)	-	DBCLOB(MIN(A+B, 1G))
BLOB(A) BLOB(B)	-	BLOB(MIN(A+B, 2G))

Note that, for compatibility with previous versions, there is no automatic escalation of results involving LONG data types to LOB data types. For example, concatenation of a CHAR(200) value and a completely full LONG VARCHAR value would result in an error rather than in a promotion to a CLOB data type.

The code page of the result is considered a derived code page and is determined by the code page of its operands as explained in “Rules for String Conversions” on page 111.

One operand may be a parameter marker. If a parameter marker is used, then the data type and length attributes of that operand are considered to be the same as those for the non-parameter marker operand. The order of operations must be considered to determine these attributes in cases with nested concatenation.

Example 1: If FIRSTNME is Pierre and LASTNAME is Fermat, then the following :

```
FIRSTNME CONCAT ' ' CONCAT LASTNAME
```

returns the value `Pierre Fermat`.

Example 2: Given:

- `COLA` defined as VARCHAR(5) with value `'AA'`
- `:host_var` defined as a character host variable with length 5 and value `'BB   '`
- `COLC` defined as CHAR(5) with value `'CC'`
- `COLD` defined as CHAR(5) with value `'DDDDD'`

The value of: `COLA` **`CONCAT`** `:host_var` **`CONCAT`** `COLC` **`CONCAT`** `COLD` is:

```
          'AABB   CC   DDDDD'
```

The data type is VARCHAR, the length attribute is 17 and the result code page is the database code page.

Example 3: Given:
  `COLA` is defined as CHAR(10)
  `COLB` is defined as VARCHAR(5)

The parameter marker in the expression:

```
  COLA CONCAT COLB CONCAT ?
```

is considered VARCHAR(15) since `COLA CONCAT COLB` is evaluated first giving a result which is the first operand of the second CONCAT operation.

User-defined Types

A user-defined type cannot be used with the concatenation operator, even if it is a distinct type with a source data type that is a string type. To concatenate, create a function with the CONCAT operator as its source. For example, if there were distinct types TITLE and TITLE_DESCRIPTION, both of which had VARCHAR(25) data types, then the following user-defined function, ATTACH, could be used to concatenate them.

```
CREATE FUNCTION ATTACH (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

Alternately, the concatenation operator could be overloaded using a user-defined function to add the new data types.

```
CREATE FUNCTION CONCAT (TITLE, TITLE_DESCRIPTION)
  RETURNS VARCHAR(50) SOURCE CONCAT (VARCHAR(), VARCHAR())
```

With Arithmetic Operators

If arithmetic operators are used, the result of the expression is a value derived from the application of the operators to the values of the operands.

If any operand can be null, or the database is configured with DFT_SQLMATHWARN set to yes, the result can be null.

If any operand has the null value, the result of the expression is the null value.

Arithmetic operators can be applied to signed numeric types and datetime types (see "Datetime Arithmetic in SQL" on page 165). For example, USER+2 is invalid. Sourced functions can be defined for arithmetic operations on distinct types with a source type that is a signed numeric type.

The prefix operator + (unary plus) does not change its operand. The prefix operator − (unary minus) reverses the sign of a nonzero operand; and if the data type of A is small integer, then the data type of −A is large integer. The first character of the token following a prefix operator must not be a plus or minus sign.

The *infix operators* +, −, *, and / specify addition, subtraction, multiplication, and division, respectively. The value of the second operand of division must not be zero. These operators can also be treated as functions. Thus, the expression *"+"(a,b)* is equivalent to the expression *a+b*. "operator" function.

Arithmetic Errors

If an arithmetic error such as zero divide or a numeric overflow occurs during the processing of an expression, an error is returned and the SQL statement processing the expression fails with an error (SQLSTATE 22003 or 22012).

A database can be configured (using DFT_SQLMATHWARN set to yes) so that arithmetic errors return a null value for the expression, issue a warning (SQLSTATE 01519 or 01564), and proceed with processing of the SQL statement. When arithmetic errors are treated as nulls, there are implications on the results of SQL statements. The following are some examples of these implications.

- An arithmetic error that occurs in the expression that is the argument of a column function causes the row to be ignored in the determining the result of the column function. If the arithmetic error was an overflow, this may significantly impact the result values.
- An arithmetic error that occurs in the expression of a predicate in a WHERE clause can cause rows to not be included in the result.
- An arithmetic error that occurs in the expression of a predicate in a check constraint results in the update or insert proceeding since the constraint is not false.

If these types of impacts are not acceptable, additional steps should be taken to handle the arithmetic error to produce acceptable results. Some examples are:

- add a case expression to check for zero divide and set the desired value for such a situation
- add additional predicates to handle nulls (like a check constraint on not nullable columns could become:

  ```
  check (c1*c2 is not null and c1*c2>5000)
  ```

  to cause the constraint to be violated on an overflow).

Two Integer Operands

If both operands of an arithmetic operator are integers, the operation is performed in binary and the result is a *large integer* unless either (or both) operand is a big integer, in which case the result is a big integer. Any remainder of division is lost. The result of an integer arithmetic operation (including unary minus) must be within the range of the result type.

Integer and Decimal Operands

If one operand is an integer and the other is a decimal, the operation is performed in decimal using a temporary copy of the integer which has been converted to a decimal number with precision *p* and scale 0. *p* is 19 for a big integer, 11 for a large integer and 5 for a small integer.

Two Decimal Operands

If both operands are decimal, the operation is performed in decimal. The result of any decimal arithmetic operation is a decimal number with a precision and scale that are dependent on the operation and the precision and scale of the operands. If the operation is addition or subtraction and the operands do not have the same scale, the operation is performed with a temporary copy of one of the operands. The copy of the shorter operand is extended with trailing zeros so that its fractional part has the same number of digits as the longer operand.

The result of a decimal operation must not have a precision greater than 31. The result of decimal addition, subtraction, and multiplication is derived from a temporary result which may have a precision greater than 31. If the precision of the temporary result is not greater than 31, the final result is the same as the temporary result.

Decimal Arithmetic in SQL

The following formulas define the precision and scale of the result of decimal operations in SQL. The symbols p and s denote the precision and scale of the first operand, and the symbols p' and s' denote the precision and scale of the second operand.

Addition and Subtraction

The precision is $\min(31,\max(p-s,p'-s') + \max(s,s')+1)$. The scale of the result of addition and subtraction is $\max(s,s')$.

Multiplication

The precision of the result of multiplication is $\min(31,p+p')$ and the scale is $\min(31,s+s')$.

Division

The precision of the result of division is 31. The scale is $31-p+s-s'$. The scale must not be negative.

Floating-Point Operands

If either operand of an arithmetic operator is floating-point, the operation is performed in floating-point, the operands having first been converted to double-precision floating-point numbers, if necessary. Thus, if any element of an expression is a floating-point number, the result of the expression is a double-precision floating-point number.

An operation involving a floating-point number and an integer is performed with a temporary copy of the integer which has been converted to double-precision floating-point. An operation involving a floating-point number and a decimal number is performed with a temporary copy of the decimal number which has been converted to double-precision floating-point. The result of a floating-point operation must be within the range of floating-point numbers.

User-defined Types as Operands

A user-defined type cannot be used with arithmetic operators even if its source data type is numeric. To perform an arithmetic operation, create a function with the arithmetic operator as its source. For example, if there were distinct types INCOME and EXPENSES, both of which had DECIMAL(8,2) data types, then the following user-defined function, REVENUE, could be used to subtract one from the other.

```
CREATE FUNCTION REVENUE (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Alternately, the - (minus) operator could be overloaded using a user-defined function to subtract the new data types.

```
CREATE FUNCTION "-" (INCOME, EXPENSES)
  RETURNS DECIMAL(8,2) SOURCE "-" (DECIMAL, DECIMAL)
```

Scalar Fullselect

A *scalar fullselect* as supported in an expression is a fullselect, enclosed in parentheses, that returns a single row consisting of a single column value. If the fullselect does not return a row, the result of the expression is the null value. If the select list element is an expression that is simply a column name or a dereference operation, the result column name is based on the name of the column. See “fullselect” on page 434 for more information.

Datetime Operations and Durations

Datetime values can be incremented, decremented, and subtracted. These operations may involve decimal numbers called *durations*. Following is a definition of durations and a specification of the rules for datetime arithmetic.

A duration is a number representing an interval of time. There are four types of durations:

Labeled Durations

labeled-duration:

```
├──┬─function──────┬──┬─YEAR─────────┬────────────────────────────────────────┤
   ├─(expression)──┤  ├─YEARS────────┤
   ├─constant──────┤  ├─MONTH────────┤
   ├─column-name───┤  ├─MONTHS───────┤
   └─host-variable─┘  ├─DAY──────────┤
                      ├─DAYS─────────┤
                      ├─HOUR─────────┤
                      ├─HOURS────────┤
                      ├─MINUTE───────┤
                      ├─MINUTES──────┤
                      ├─SECOND───────┤
                      ├─SECONDS──────┤
                      ├─MICROSECOND──┤
                      └─MICROSECONDS─┘
```

A *labeled duration* represents a specific unit of time as expressed by a number (which can be the result of an expression) followed by one of the seven duration keywords: YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS, or MICROSECONDS.[33] The number specified is converted as if it were assigned to a DECIMAL(15,0) number. A labeled duration can only be used as an operand of an arithmetic operator in which the other operand is a value of data type DATE, TIME, or TIMESTAMP. Thus, the expression HIREDATE + 2 MONTHS + 14 DAYS is valid, whereas the expression HIREDATE + (2 MONTHS + 14 DAYS) is not. In both of these expressions, the labeled durations are 2 MONTHS and 14 DAYS.

Date Duration

A *date duration* represents a number of years, months, and days, expressed as a DECIMAL(8,0) number. To be properly interpreted, the number must have the format *yyyymmdd.*, where *yyyy* represents the number of years, *mm* the number of months, and *dd* the number of days. [34] The result of subtracting one date value from another, as in the expression HIREDATE – BRTHDATE, is a date duration.

Time Duration

A *time duration* represents a number of hours, minutes, and seconds, expressed as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format *hhmmss.*, where *hh* represents the number of hours, *mm* the number of minutes, and *ss* the number of seconds. [34] The result of subtracting one time value from another is a time duration.

Timestamp duration

A *timestamp duration* represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the number must have the format *yyyymmddhhmmss.zzzzzz*, where *yyyy*, *mm*, *dd*, *hh*, *mm*, *ss*, and *zzzzzz* represent, respectively, the number of years, months, days, hours, minutes, seconds, and microseconds. The result of subtracting one timestamp value from another is a timestamp duration.

Datetime Arithmetic in SQL

The only arithmetic operations that can be performed on datetime values are addition and subtraction. If a datetime value is the operand of addition, the other operand must be a duration. The specific rules governing the use of the addition operator with datetime values follow.

- If one operand is a date, the other operand must be a date duration or labeled duration of YEARS, MONTHS, or DAYS.
- If one operand is a time, the other operand must be a time duration or a labeled duration of HOURS, MINUTES, or SECONDS.

33. Note that the singular form of these keywords is also acceptable: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and MICROSECOND.

34. The period in the format indicates a DECIMAL data type.

- If one operand is a timestamp, the other operand must be a duration. Any type of duration is valid.
- Neither operand of the addition operator can be a parameter marker.

The rules for the use of the subtraction operator on datetime values are not the same as those for addition because a datetime value cannot be subtracted from a duration, and because the operation of subtracting two datetime values is not the same as the operation of subtracting a duration from a datetime value. The specific rules governing the use of the subtraction operator with datetime values follow.

- If the first operand is a date, the second operand must be a date, a date duration, a string representation of a date, or a labeled duration of YEARS, MONTHS, or DAYS.
- If the second operand is a date, the first operand must be a date, or a string representation of a date.
- If the first operand is a time, the second operand must be a time, a time duration, a string representation of a time, or a labeled duration of HOURS, MINUTES, or SECONDS.
- If the second operand is a time, the first operand must be a time, or string representation of a time.
- If the first operand is a timestamp, the second operand must be a timestamp, a string representation of a timestamp, or a duration.
- If the second operand is a timestamp, the first operand must be a timestamp or a string representation of a timestamp.
- Neither operand of the subtraction operator can be a parameter marker.

Date Arithmetic

Dates can be subtracted, incremented, or decremented.

Subtracting Dates: The result of subtracting one date (DATE2) from another (DATE1) is a date duration that specifies the number of years, months, and days between the two dates. The data type of the result is DECIMAL(8,0). If DATE1 is greater than or equal to DATE2, DATE2 is subtracted from DATE1. If DATE1 is less than DATE2, however, DATE1 is subtracted from DATE2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = DATE1 − DATE2.

```
If DAY(DATE2) <= DAY(DATE1)
     then DAY(RESULT) = DAY(DATE1) − DAY(DATE2).

If DAY(DATE2) > DAY(DATE1)
     then DAY(RESULT) = N + DAY(DATE1) − DAY(DATE2)
        where N = the last day of MONTH(DATE2).
           MONTH(DATE2) is then incremented by 1.

If MONTH(DATE2) <= MONTH(DATE1)
     then MONTH(RESULT) = MONTH(DATE1) - MONTH(DATE2).
```

```
If MONTH(DATE2) > MONTH(DATE1)
     then MONTH(RESULT) = 12 + MONTH(DATE1) - MONTH(DATE2).
        YEAR(DATE2) is then incremented by 1.
YEAR(RESULT) = YEAR(DATE1) - YEAR(DATE2).
```

For example, the result of DATE('3/15/2000') – '12/31/1999' is 00000215. (or, a duration of 0 years, 2 months, and 15 days).

Incrementing and Decrementing Dates: The result of adding a duration to a date, or of subtracting a duration from a date, is itself a date. (For the purposes of this operation, a month denotes the equivalent of a calendar page. Adding months to a date, then, is like turning the pages of a calendar, starting with the page on which the date appears.) The result must fall between the dates January 1, 0001 and December 31, 9999 inclusive.

If a duration of years is added or subtracted, only the year portion of the date is affected. The month is unchanged, as is the day unless the result would be February 29 of a non-leap-year. In this case, the day is changed to 28, and a warning indicator in the SQLCA is set to indicate the adjustment.

Similarly, if a duration of months is added or subtracted, only months and, if necessary, years are affected. The day portion of the date is unchanged unless the result would be invalid (September 31, for example). In this case, the day is set to the last day of the month, and a warning indicator in the SQLCA is set to indicate the adjustment.

Adding or subtracting a duration of days will, of course, affect the day portion of the date, and potentially the month and year.

Date durations, whether positive or negative, may also be added to and subtracted from dates. As with labeled durations, the result is a valid date, and a warning indicator is set in the SQLCA whenever an end-of-month adjustment is necessary.

When a positive date duration is added to a date, or a negative date duration is subtracted from a date, the date is incremented by the specified number of years, months, and days, in that order. Thus, DATE1 + X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 + YEAR(X) YEARS + MONTH(X) MONTHS + DAY(X) DAYS.
```

When a positive date duration is subtracted from a date, or a negative date duration is added to a date, the date is decremented by the specified number of days, months, and years, in that order. Thus, DATE1 – X, where X is a positive DECIMAL(8,0) number, is equivalent to the expression:

```
DATE1 - DAY(X) DAYS - MONTH(X) MONTHS - YEAR(X) YEARS.
```

When adding durations to dates, adding one month to a given date gives the same date one month later unless that date does not exist in the later month. In that case, the date is set to that of the last day of the later month. For example, January 28 plus one month gives February 28; and one month added to January 29, 30, or 31 results in either February 28 or, for a leap year, February 29.

Note: If one or more months is added to a given date and then the same number of months is subtracted from the result, the final date is not necessarily the same as the original date.

Time Arithmetic

Times can be subtracted, incremented, or decremented.

Subtracting Times: The result of subtracting one time (TIME2) from another (TIME1) is a time duration that specifies the number of hours, minutes, and seconds between the two times. The data type of the result is DECIMAL(6,0).

If TIME1 is greater than or equal to TIME2, TIME2 is subtracted from TIME1.

If TIME1 is less than TIME2, however, TIME1 is subtracted from TIME2, and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TIME1 – TIME2.

```
If SECOND(TIME2) <= SECOND(TIME1)
     then SECOND(RESULT) = SECOND(TIME1) - SECOND(TIME2).

If SECOND(TIME2) > SECOND(TIME1)
     then SECOND(RESULT) = 60 + SECOND(TIME1) - SECOND(TIME2).
        MINUTE(TIME2) is then incremented by 1.

If MINUTE(TIME2) <= MINUTE(TIME1)
     then MINUTE(RESULT) = MINUTE(TIME1) - MINUTE(TIME2).

If MINUTE(TIME1) > MINUTE(TIME1)
     then MINUTE(RESULT) = 60 + MINUTE(TIME1) - MINUTE(TIME2).
        HOUR(TIME2) is then incremented by 1.

HOUR(RESULT) = HOUR(TIME1) - HOUR(TIME2).
```

For example, the result of TIME(’11:02:26’) – ’00:32:56’ is 102930. (a duration of 10 hours, 29 minutes, and 30 seconds).

Incrementing and Decrementing Times: The result of adding a duration to a time, or of subtracting a duration from a time, is itself a time. Any overflow or underflow of hours is discarded, thereby ensuring that the result is always a time. If a duration of hours is added or subtracted, only the hours portion of the time is affected. The minutes and seconds are unchanged.

Similarly, if a duration of minutes is added or subtracted, only minutes and, if necessary, hours are affected. The seconds portion of the time is unchanged.

Adding or subtracting a duration of seconds will, of course, affect the seconds portion of the time, and potentially the minutes and hours.

Time durations, whether positive or negative, also can be added to and subtracted from times. The result is a time that has been incremented or decremented by the specified number of hours, minutes, and seconds, in that order. `TIME1 + X`, where “X” is a DECIMAL(6,0) number, is equivalent to the expression:

```
TIME1 + HOUR(X) HOURS + MINUTE(X) MINUTES + SECOND(X) SECONDS
```

Note: Although the time ’24:00:00’ is accepted as a valid time, it is never returned as the result of time addition or subtraction, even if the duration operand is zero (e.g. time(’24:00:00’)±0 seconds = ’00:00:00’).

Timestamp Arithmetic

Timestamps can be subtracted, incremented, or decremented.

Subtracting Timestamps: The result of subtracting one timestamp (TS2) from another (TS1) is a timestamp duration that specifies the number of years, months, days, hours, minutes, seconds, and microseconds between the two timestamps. The data type of the result is DECIMAL(20,6).

If TS1 is greater than or equal to TS2, TS2 is subtracted from TS1. If TS1 is less than TS2, however, TS1 is subtracted from TS2 and the sign of the result is made negative. The following procedural description clarifies the steps involved in the operation result = TS1 − TS2:

```
If MICROSECOND(TS2) <= MICROSECOND(TS1)
     then MICROSECOND(RESULT) = MICROSECOND(TS1) -
        MICROSECOND(TS2).
If MICROSECOND(TS2) > MICROSECOND(TS1)
     then MICROSECOND(RESULT) = 1000000 +
        MICROSECOND(TS1) - MICROSECOND(TS2)
           and SECOND(TS2) is incremented by 1.
```

The seconds and minutes part of the timestamps are subtracted as specified in the rules for subtracting times.

```
If HOUR(TS2) <= HOUR(TS1)
     then HOUR(RESULT) = HOUR(TS1) - HOUR(TS2).
If HOUR(TS2) > HOUR(TS1)
     then HOUR(RESULT) = 24 + HOUR(TS1) - HOUR(TS2)
        and DAY(TS2) is incremented by 1.
```

The date part of the timestamps is subtracted as specified in the rules for subtracting dates.

Incrementing and Decrementing Timestamps: The result of adding a duration to a timestamp, or of subtracting a duration from a timestamp is itself a timestamp. Date and time arithmetic is performed as previously defined, except that an overflow or underflow of hours is carried into the date part of the result, which must be within the range of valid dates. Microseconds overflow into seconds.

Precedence of Operations

Expressions within parentheses and dereference operations are evaluated first from left to right. [35] When the order of evaluation is not specified by parentheses, prefix operators are applied before multiplication and division, and multiplication and division are applied before addition and subtraction. Operators at the same precedence level are applied from left to right.

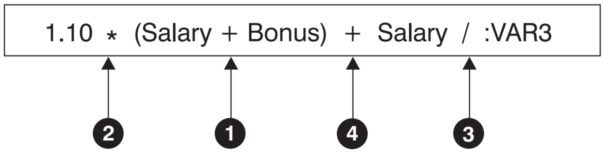


Figure 11. Precedence of Operations

35. Note that parentheses are also used in subselect statements, search conditions, and functions. However, they should not be used to arbitrarily group sections within SQL statements.

CASE Expressions

case-expression:

```
                                          ┌─ELSE NULL──────────────────┐
├──CASE──┬─searched-when-clause─┬──────────┼────────────────────────────┼─END──────────────┤
         └─simple-when-clause───┘          └─ELSE──result-expression────┘
```

searched-when-clause:

```
   ┌───────────────────────────────────────────────────────────┐
   ▼                                                            │
├────WHEN──search-condition──THEN──┬─result-expression─┬───────┴────────────────────────┤
                                   └─NULL──────────────┘
```

simple-when-clause:

```
                 ┌───────────────────────────────────────────────────────────┐
                 ▼                                                            │
├──expression──────WHEN──expression──THEN──┬─result-expression─┬────────────┴──────────────┤
                                           └─NULL──────────────┘
```

CASE expressions allow an expression to be selected based on the evaluation of one or more conditions. In general, the value of the case-expression is the value of the *result-expression* following the first (leftmost) case that evaluates to true. If no case evaluates to true and the ELSE keyword is present then the result is the value of the *result-expression* or NULL. If no case evaluates to true and the ELSE keyword is not present then the result is NULL. Note that when a case evaluates to unknown (because of NULLs), the case is not true and hence is treated the same way as a case that evaluates to false.

If the CASE expression is in a VALUES clause, an IN predicate, a GROUP BY clause, or an ORDER BY clause, the *search-condition* in a searched-when-clause cannot be a quantified predicate, IN predicate using a fullselect, or an EXISTS predicate (SQLSTATE 42625).

When using the *simple-when-clause*, the value of the *expression* prior to the first *WHEN* keyword is tested for equality with the value of the *expression* following the WHEN keyword. The data type of the *expression* prior to the first *WHEN* keyword must therefore be comparable to the data types of each *expression* following the WHEN keyword(s). The *expression* prior to the first *WHEN* keyword in a *simple-when-clause* cannot include a function that is variant or has an external action (SQLSTATE 42845).

A *result-expression* is an *expression* following the THEN or ELSE keywords. There must be at least one *result-expression* in the CASE expression (NULL cannot be specified for every case) (SQLSTATE 42625). All *result-expression*s must have compatible data types (SQLSTATE 42804), where the attributes of the result are determined based on the "Rules for Result Data Types" on page 107.

Examples:

- If the first character of a department number is a division in the organization, then a CASE expression can be used to list the full name of the division to which each employee belongs:

```
SELECT EMPNO, LASTNAME,
       CASE SUBSTR(WORKDEPT,1,1)
       WHEN 'A' THEN 'Administration'
       WHEN 'B' THEN 'Human Resources'
       WHEN 'C' THEN 'Accounting'
       WHEN 'D' THEN 'Design'
       WHEN 'E' THEN 'Operations'
       END
FROM EMPLOYEE;
```

- The number of years of education are used in the EMPLOYEE table to give the education level. A CASE expression can be used to group these and to show the level of education.

```
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
       CASE
       WHEN EDLEVEL < 15 THEN 'SECONDARY'
       WHEN EDLEVEL < 19 THEN 'COLLEGE'
       ELSE 'POST GRADUATE'
       END
FROM EMPLOYEE
```

- Another interesting example of CASE statement usage is in protecting from division by 0 errors. For example, the following code finds the employees who earn more than 25% of their income from commission, but who are not fully paid on commission:

```
SELECT EMPNO, WORKDEPT, SALARY+COMM FROM EMPLOYEE
WHERE (CASE WHEN SALARY=0 THEN NULL
            ELSE COMM/SALARY
            END) > 0.25;
```

- The following CASE expressions are the same:

```
SELECT LASTNAME,
 CASE
 WHEN LASTNAME = 'Haas' THEN 'President'
 ...
```

```
SELECT LASTNAME,
 CASE LASTNAME
 WHEN 'Haas' THEN 'President'
 ...
```

There are two scalar functions, NULLIF and COALESCE, that are specialized to handle a subset of the functionality provided by CASE. Table 11 shows the equivalent expressions using CASE or these functions.

Table 11. Equivalent CASE Expressions

Expression	Equivalent Expression
CASE WHEN e1=e2 THEN NULL ELSE e1 END	NULLIF(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE e2 END	COALESCE(e1,e2)
CASE WHEN e1 IS NOT NULL THEN e1 ELSE COALESCE(e2,...,eN) END	COALESCE(e1,e2,...,eN)

CAST Specifications

cast-specification:

```
├──CAST──(──┬─expression───────┬──AS──data-type──────────────────────────────►
            ├─NULL─────────────┤
            └─parameter-marker─┘

►──┬──────────────────────────────────────┬──)────────────────────────────┤
   │        (1)                           │
   └─SCOPE──────────┬─typed-table-name─┬──┘
                    └─typed-view-name──┘
```

Notes:

1 The SCOPE clause only applies to the REF data type.

The CAST specification returns the cast operand (the first operand) cast to the type specified by the *data type*.

expression

If the cast operand is an expression (other than parameter marker or NULL), the result is the argument value converted to the specified target *data type*.

The supported casts are shown in Table 6 on page 93 where the first column represents the data type of the cast operand (source data type) and the data types across the top represent the target data type of the CAST specification. If the cast is not supported an error will occur (SQLSTATE 42846).

When casting character strings (other than CLOBs) to a character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. When casting graphic character strings (other than DBCLOBs) to a graphic character string with a different length, a warning (SQLSTATE 01004) is returned if truncation of other than trailing blanks occurs. For BLOB, CLOB and DBCLOB cast operands, the warning is issued if any characters are truncated.

NULL
If the cast operand is the keyword NULL, the result is a null value that has the specified *data type*.

parameter-marker
A parameter marker (specified as a question mark character) is normally considered an expression, but is documented separately in this case because it has a special meaning. If the cast operand is a *parameter-marker*, the specified *data type* is considered a promise that the replacement will be assignable to the specified data type (using store assignment for strings). Such a parameter marker is considered a *typed parameter marker*. Typed parameter markers will be treated like any other typed value for the purpose of function resolution, DESCRIBE of a select list or for column assignment.

data type
The name of an existing data type. If the type name is not qualified, the SQL path is used to do data type resolution. A data type that has an associated attributes like length or precision and scale should include these attributes when specifying *data type* (CHAR defaults to a length of 1 and DECIMAL defaults to a precision of 5 and scale of 0 if not specified). Restrictions on the supported data types are based on the specified cast operand.

- For a cast operand that is an *expression*, see “Casting Between Data Types” on page 91 for the target data types that are supported based on the data type of the cast operand (source data type).
- For a cast operand that is the keyword NULL, any existing data type can be used.
- For a cast operand that is a parameter marker, the target data type can be any existing data type. If the data type is a user-defined distinct type, the application using the parameter marker will use the source data type of the user-defined distinct type. If the data type is a user-defined structured type, the application using the parameter marker will use the input parameter type of the TO SQL transform function for the user-defined structured type.

SCOPE
When the data type is a reference type, a scope may be defined that identifies the target table or target view of the reference.

typed-table-name
The name of a typed table. The table must already exist (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-table-name* (SQLSTATE 428DM).

typed-view-name
: The name of a typed view. The view must exist or have the same name as the view being created that includes the cast as part of the view definition (SQLSTATE 42704). The cast must be to *data-type* REF(*S*), where *S* is the type of *typed-view-name* (SQLSTATE 428DM).

When numeric data is cast to character the result data type is a fixed-length character string (see "CHAR" on page 260). When character data is cast to numeric, the result data type depends on the type of number specified. For example, if cast to integer, it would become a large integer (see "INTEGER" on page 310).

Examples:

- An application is only interested in the integer portion of the SALARY (defined as decimal(9,2)) from the EMPLOYEE table. The following query, including the employee number and the integer value of SALARY, could be prepared.

  ```
  SELECT EMPNO, CAST(SALARY AS INTEGER) FROM EMPLOYEE
  ```
- Assume the existence of a distinct type called T_AGE that is defined on SMALLINT and used to create column AGE in PERSONNEL table. Also assume the existence of a distinct type called R_YEAR that is defined on INTEGER and used to create column RETIRE_YEAR in PERSONNEL table. The following update statement could be prepared.

  ```
  UPDATE PERSONNEL SET RETIRE_YEAR =?
                WHERE AGE = CAST( ? AS T_AGE)
  ```

  The first parameter is an untyped parameter marker that would have a data type of R_YEAR, although the application will use an integer for this parameter marker. This does not require the explicit CAST specification because it is an assignment.

  The second parameter marker is a typed parameter marker that is cast as a distinct type T_AGE. This satisfies the requirement that the comparison must be performed with compatible data types. The application will use the source data type (which is SMALLINT) for processing this parameter marker.

  Successful processing of this statement assumes that the function path includes the schema name of the schema (or schemas) where the two distinct types are defined.

Dereference Operations

dereference-operation:

```
├──scoped-ref-expression── -> ──name1──┬──────────────────────────────┬──────────┤
                                        └─(──┬────────────────────┬──)──┘
                                             │  ┌─,───────────┐   │
                                             │  ▼             │   │
                                             └───expression───┴───┘
```

The scope of the scoped reference expression is a table or view called the *target* table or view. The scoped reference expression identifies a *target row*. The *target row* is the row in the target table or view (or in one of its subtables or subviews) whose object identifier (OID) column value matches the reference expression. See "CREATE TABLE" on page 712 for further information about OID columns. The dereference operation can be used to access a column of the target row, or to invoke a method, using the target row as the subject of the method. The result of a dereference operation can always be null. The dereference operation takes precedence over all other operators.

scoped-ref-expression
An expression that is a reference type that has a scope (SQLSTATE 428DT). If the expression is a host variable, parameter marker or other unscoped reference type value, a CAST specification with a SCOPE clause is required to give the reference a scope.

name1
Specifies an unqualified identifier.

If no parentheses follow *name1*, and *name1* matches the name of an attribute of the target type, then the value of the dereference operation is the value of the named column in the target row. In this case, the data type of the column (made nullable) determines the result type of the dereference operation. If no target row exists whose object identifier matches the reference expression, then the result of the dereference operation is null. If the dereference operation is used in a select list and is not included as part of an expression, *name1* becomes the result column name.

If parentheses follow *name1*, or if *name1* does not match the name of an attribute of the target type, then the dereference operation is treated as a method invocation. The name of the invoked method is *name1*. The subject of the method is the target row, considered as an instance of its structured type. If no target row exists whose object identifier matches the reference expression, the subject of the method is a null value of the target type. The expressions inside parentheses, if any, provide the remaining parameters of the method invocation. The normal process is used for resolution of the method invocation. The result type of the selected method (made nullable) determines the result type of the dereference operation.

The authorization ID of the statement that uses a dereference operation must have SELECT privilege on the target table of the *scoped-ref-expression* (SQLSTATE 42501).

A dereference operation can never modify values in the database. If a dereference operation is used to invoke a mutator method, the mutator method modifies a copy of the target row and returns the copy, leaving the database unchanged.

Examples:

- Assume the existence of an EMPLOYEE table that contains a column called DEPTREF which is a reference type scoped to a typed table based on a type that includes the attribute DEPTNAME. The values of DEPTREF in the table EMPLOYEE should correspond to the OID column values in the target table of DEPTREF column.

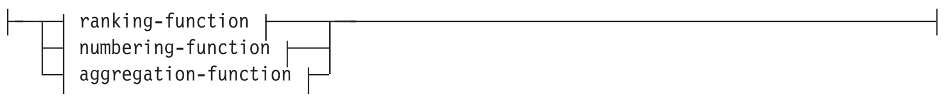
```
SELECT EMPNO, DEPTREF->DEPTNAME
  FROM EMPLOYEE
```

- Using the same tables as in the previous example, use a dereference operation to invoke a method named BUDGET, with the target row as subject parameter, and '1997' as an additional parameter.

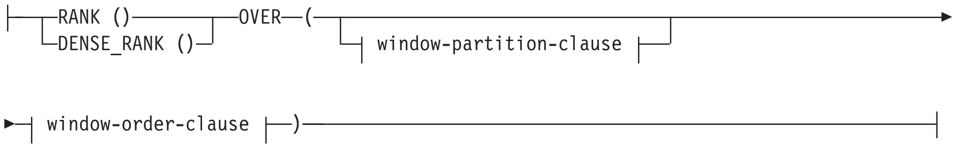
```
SELECT EMPNO, DEPTREF->BUDGET('1997')
  AS DEPTBUDGET97
  FROM EMPLOYEE
```

OLAP Functions

OLAP-function:



ranking-function:



numbering-function:

```
├──ROW_NUMBER ()──OVER──(─┬────────────────────────────┬──────────────►
                          └─┤ window-partition-clause ├─┘

►─┬────────────────────────┬─)──────────────────────────────────────┤
  └─┤ window-order-clause ├─┘
```

aggregation-function:

```
├──column-function──OVER──(─┬────────────────────────────┬─────────────►
                            └─┤ window-partition-clause ├─┘

►─┬────────────────────────┬───────────────────────────────────────────►
  └─┤ window-order-clause ├─┘

  ┌─RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING──┐
►─┼────────────────────────────────────────────────────────────┼─)──────┤
  └─┤ window-aggregation-group-clause ├────────────────────────┘
```

window-partition-clause:

```
                 ┌─,──────────────────────┐
                 ▼                        │
├──PARTITION BY────partitioning-expression─┴──────────────────────────┤
```

window-order-clause:

```
             ┌─,────────────────────────────────┐
             │                       ┌─ASC──┐   │
             ▼                       │      │   │
├──ORDER BY────sort-key-expression───┤      ├───┴────────────────────┤
                                     └─DESC─┘
```

window-aggregation-group-clause:

```
├───┬─ROWS──┬──┬─┤ group-start ├───┬──────────────────────────────────┤
    └─RANGE─┘  └─┤ group-between ├─┘
```

group-start:

```
├──┬─UNBOUNDED PRECEDING──────────┬────────────────────────────────────────────┤
   ├─unsigned-constant──PRECEDING─┤
   └─CURRENT ROW──────────────────┘
```

group-between:

```
├──BETWEEN──┤ group-bound1 ├──AND──┤ group-bound2 ├──────────────────────────────┤
```

group-bound1:

```
├──┬─UNBOUNDED PRECEDING──────────┬────────────────────────────────────────────┤
   ├─unsigned-constant──PRECEDING─┤
   ├─unsigned-constant──FOLLOWING─┤
   └─CURRENT ROW──────────────────┘
```

group-bound2:

```
├──┬─UNBOUNDED FOLLOWING──────────┬────────────────────────────────────────────┤
   ├─unsigned-constant──PRECEDING─┤
   ├─unsigned-constant──FOLLOWING─┤
   └─CURRENT ROW──────────────────┘
```

On-Line Analytical Processing (OLAP) functions provide the ability to return ranking, row numbering and existing column function information as a scalar value in a query result. An OLAP function can be included in expressions in a select-list or the ORDER BY clause of a select-statement (SQLSTATE 42903). An OLAP function cannot be used as an argument of a column function (SQLSTATE 42607). The query result to which the OLAP function is applied is the result table of the innermost subselect that includes the OLAP function.

When specifying an OLAP function, a window is specified that defines the rows over which the function is applied, and in what order. When used with a column function, the applicable rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three month period.

The ranking function computes the ordinal rank of a row within the window. Rows that are not distinct with respect to the ordering within their window are assigned the same rank. The results of ranking may be defined with or without gaps in the numbers resulting from duplicate values.

If RANK is specified, the rank of a row is defined as 1 plus the number of rows that strictly precede the row. Thus, if two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering.

If DENSE_RANK[36] is specified, the rank of a row is defined as 1 plus the number of rows preceding that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering.

The ROW_NUMBER[37] function computes the sequential row number of the row within the window defined by the ordering, starting with 1 for the first row. If the ORDER BY clause is not specified in the window, the row numbers are assigned to the rows in arbitrary order as returned by the subselect (not according to any ORDER BY clause in the select-statement).

The data type of the result of RANK, DENSE_RANK or ROW_NUMBER is BIGINT. The result cannot be null.

PARTITION BY (*partitioning-expression***,...)**
Defines the partition within which the function is applied. A *partitioning-expression* is an expression used in defining the partitioning of the result set. Each *column-name* referenced in a partitioning-expression must unambiguously reference a result set column of the OLAP function subselect statement (SQLSTATE 42702 or 42703). The length of each partitioning-expression must not be more than 255 bytes (SQLSTATE 42907). A partitioning-expression cannot include a scalar-fullselect (SQLSTATE 42822) or any function that is not deterministic or has an external action (SQLSTATE 42845).

ORDER BY (*sort-key-expression***,...)**
Defines the ordering of rows within a partition that determine the value of the OLAP function or the meaning of the ROW values in the window-aggregation-group-clause (it does not define the ordering of the query result set). A *sort-key-expression* is an expression used in defining the ordering of the rows within a window partition. Each column-name referenced in a sort-key-expression must unambiguously reference a column of the result set of the subselect including the OLAP function (SQLSTATE 42702 or 42703). The length of each sort-key-expression must not be more than 255 bytes (SQLSTATE 42907). A sort-key-expression cannot include a scalar fullselect (SQLSTATE 42822) or any function that is not deterministic or has an external action (SQLSTATE 42845). This clause is required for the RANK and DENSE_RANK functions (SQLSTATE 42601).

36. DENSE_RANK and DENSERANK are synonyms.

37. ROW_NUMBER and ROWNUMBER are synonyms.

ASC
Uses the values of the sort-key-expression in ascending order. Null values are considered last in the order.

DESC
Uses the values of the sort-key-expression in descending order. Null values are considered first in the order.

window-aggregation-group-clause
The aggregation group of a row R is a set of rows, defined relative to R in the ordering of the rows of R's partition. This clause specifies the aggregation group.

ROWS
Indicates the aggregation group is defined by counting rows.

RANGE
Indicates the aggregation group is defined by an offset from a sort key.

group-start
Specifies the starting point for the aggregation group. The aggregation group end is the current row. Specification of the group-start clause is equivalent to a group-between clause of the form "BETWEEN group-start AND CURRENT ROW".

group-between
Specifies the aggregation group start and end based on either ROWS or RANGE.

UNBOUNDED PRECEDING
Includes the entire partition preceding the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

UNBOUNDED FOLLOWING
Includes the entire partition following the current row. This can be specified with either ROWS or RANGE. Also, this can be specified with multiple sort-key-expressions in the window-order-clause.

CURRENT ROW
Specifies the start or end of the aggregation group as the current row. This clause cannot be specified in *group-bound2* if *group-bound1* specifies *value* FOLLOWING.

value **PRECEDING**
Specifies either the range or number of rows preceding the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and

the data type of the sort-key-expression must allow subtraction. This clause cannot be specified in *group-bound2* if *group-bound1* is CURRENT ROW or *value* FOLLOWING.

value **FOLLOWING**
Specifies either the range or number of rows following the current row. If ROWS is specified, then *value* is a positive integer indicating a number of rows. If RANGE is specified, then the data type of *value* must be comparable to the type of the sort-key-expression of the window-order-clause. There can only be one sort-key-expression, and the data type of the sort-key-expression must allow addition.

Examples:

- Display the ranking of employees, in order by surname, according to their total salary (based on salary plus bonus) that have a total salary more than $30,000.

```
SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
       RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
   FROM EMPLOYEE WHERE SALARY+BONUS > 30000
   ORDER BY LASTNAME
```

  Note that if the result is to be ordered by the ranking, then replace ORDER BY LASTNAME with:

```
ORDER BY RANK_SALARY
```

  or

```
ORDER BY RANK() OVER (ORDER BY SALARY+BONUS DESC)
```

- Rank the departments according to their average total salary.

```
SELECT WORKDEPT, AVG(SALARY+BONUS) AS AVG_TOTAL_SALARY,
       RANK() OVER (ORDER BY AVG(SALARY+BONUS) DESC) AS RANK_AVG_SAL
   FROM EMPLOYEE
   GROUP BY WORKDEPT
   ORDER BY RANK_AVG_SAL
```

- Rank the employees within a department according to their education level. Having multiple employees with the same rank in the department should not increase the next ranking value.

```
SELECT WORKDEPT, EMPNO, LASTNAME, FIRSTNME, EDLEVEL
       DENSE_RANK() OVER
         (PARTITION BY WORKDEPT ORDER BY EDLEVEL DESC) AS RANK_EDLEVEL
   FROM EMPLOYEE
   ORDER BY WORKDEPT, LASTNAME
```

- Provide row numbers in the result of a query.

```
SELECT ROW_NUMBER() OVER (ORDER BY WORKDEPT, LASTNAME) AS NUMBER,
       LASTNAME, SALARY
  FROM EMPLOYEE
  ORDER BY WORKDEPT, LASTNAME
```

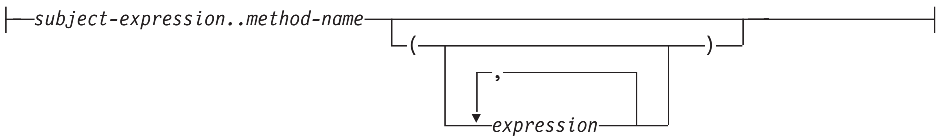
- List the top five wage earners.

```
SELECT EMPNO, LASTNAME, FIRSTNME, TOTAL_SALARY, RANK_SALARY
   FROM (SELECT EMPNO, LASTNAME, FIRSTNME, SALARY+BONUS AS TOTAL_SALARY,
                RANK() OVER (ORDER BY SALARY+BONUS DESC) AS RANK_SALARY
                FROM EMPLOYEE) AS RANKED_EMPLOYEE
   WHERE RANK_SALARY < 6
   ORDER BY RANK_SALARY
```

Notice that a nested table expression was used to first compute the result, including the rankings, before the rank could be used in the WHERE clause. A common table expression could also have been used.

Method Invocation

method-invocation:



Both system-generated observer and mutator methods, as well as user-defined methods are invoked using the double-dot operator.

subject-expression
An expression with a static result type that is a user-defined structured type.

method-name
The unqualified name of a method. The static type of *subject-expression* or one of its supertypes must include a method with the specified name.

(*expression***,...)**
The arguments of *method-name* are specified within parentheses. Empty parentheses can be used to indicate that there are no arguments. The *method-name* and the data types of the specified argument expressions are used to resolve to the specific method, based on the static type of *subject-expression* (see “Method Resolution” on page 151 for more information).

The double-dot operator used for method invocation is a high precedence left to right infix operator. For example, the following two expressions are equivalent:

```
a..b..c + x..y..z
```

and

```
((a..b)..c) + ((x..y)..z)
```

If a method has no parameters other than its subject, it may be invoked with or without parentheses. For example, the following two expressions are equivalent:

```
point1..x
```

and

```
point1..x()
```

Null subjects in method calls are handled as follows:

1. If a system-generated mutator method is invoked with a null subject, an error results (SQLSTATE 2202D)
2. If any method other than a system-generated mutator is invoked with a null subject, the method is not executed, and its result is null. This rule includes user-defined methods with SELF AS RESULT.

When a database object (a package, view, or trigger, for example) is created, the best fit method that exists for each of its method invocations is found using the rules specified in "Method Resolution" on page 151.

Note:

- Methods of types defined WITH FUNCTION ACCESS can also be invoked using the regular function notation. Function resolution considers all functions, as well as methods with function access as candidate functions. However, functions cannot be invoked using method invocation. Method resolution considers all methods and does not consider functions as candidate methods. Failure to resolve to an appropriate function or method results in an error (SQLSTATE 42884).

Examples:

- Use the double-dot operator to invoke a method called AREA. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that the method AREA has been defined previously for the CIRCLE type as `AREA() RETURNS DOUBLE`.

```
SELECT CIRCLE_COL..AREA()
  FROM RINGS
```

Subtype Treatment

subtype-treatment:

```
├──TREAT──(──expression──AS──data-type──)─────────────────────────────────┤
```

The *subtype-treatment* is used to cast a structured type expression into one of its subtypes. The static type of *expression* must be a user-defined structured type, and that type must be the same type as, or a supertype of, *data-type*. If the type name in *data-type* is unqualified, the SQL path is used to resolve the type reference. The static type of the result of subtype-treatment is *data-type*, and the value of the subtype-treatment is the value of the expression. At run time, if the dynamic type of the expression is not *data-type* or a subtype of *data-type*, an error is returned (SQLSTATE 0D000).

Examples:

- If an application knows that all column object instances in a column CIRCLE_COL have the dynamic type COLOREDCIRCLE, use the following query to invoke the method RGB on such objects. Assume the existence of a table called RINGS, with a column CIRCLE_COL of structured type CIRCLE. Also, assume that COLOREDCIRCLE is a subtype of CIRCLE and that the method RGB has been defined previously for COLOREDCIRCLE as `RGB() RETURNS DOUBLE`.

  ```
  SELECT TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
     FROM RINGS
  ```

  At run-time, if there are instances of dynamic type CIRCLE, an error is raised (SQLSTATE 0D000). This error can be avoided by using the TYPE predicate in a CASE expression, as follows:

  ```
  SELECT (CASE
     WHEN CIRCLE_COL IS OF (COLOREDCIRCLE)
        THEN TREAT (CIRCLE_COL AS COLOREDCIRCLE)..RGB()
        ELSE NULL
     END)
     FROM RINGS
  ```

  See "TYPE Predicate" on page 203 for more information.

Predicates

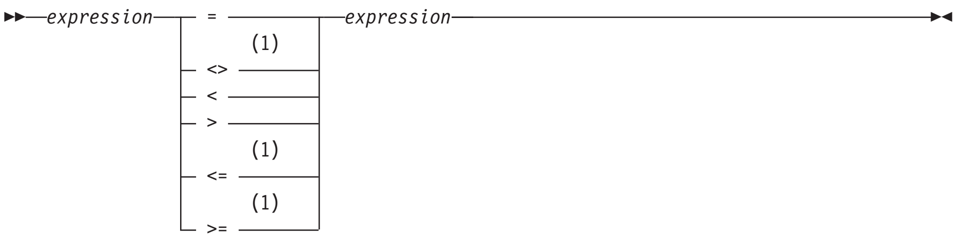
A *predicate* specifies a condition that is true, false, or unknown about a given row or group.

The following rules apply to all types of predicates:

- All values specified in a predicate must be compatible.
- An expression used in a Basic, Quantified, IN, or BETWEEN predicate must not result in a character string with a length attribute greater than 4 000, a graphic string with a length attribute greater than 2 000, or a LOB string of any size.
- The value of a host variable may be null (that is, the variable may have a negative indicator variable).
- The code page conversion of operands of predicates involving two or more operands, with the exception of LIKE, are done according to "Rules for String Conversions" on page 111
- Use of a DATALINK value is limited to the NULL predicate.
- Use of a structured type value is limited to the NULL predicate and the TYPE predicate.

A fullselect is a form of the SELECT statement which is described under "Chapter 5. Queries" on page 393. A fullselect used in a predicate is also called a *subquery*.

Basic Predicate



Notes:

1 Other comparison operators are also supported [38]

A *basic predicate* compares two values.

If the value of either operand is null, the result of the predicate is unknown. Otherwise the result is either true or false.

For values *x* and *y*:

Predicate	**Is True If and Only If...**
x **=** *y*	*x* is equal to *y*
x **<>** *y*	*x* is not equal to *y*
x **<** *y*	*x* is less than *y*
x **>** *y*	*x* is greater than *y*
x **>=** *y*	*x* is greater than or equal to *y*
x **<=** *y*	*x* is less than or equal to *y*

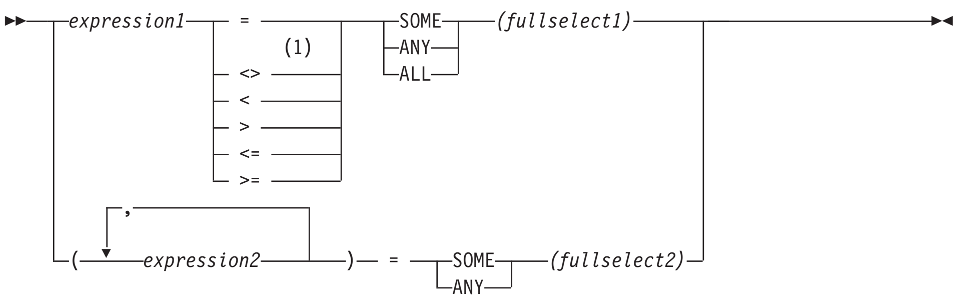
Examples:

```
EMPNO='528671'
SALARY < 20000
PRSTAFF <> :VAR1
SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE)
```

38. The following forms of the comparison operators are also supported in basic and quantified predicates; ˆ=, ˆ<, ˆ>, !=, !< and !>. In addition, in code pages 437, 819, and 850, the forms ¬=, ¬<, and ¬> are supported.

All these product-specific forms of the comparison operators are intended only to support existing SQL that uses these operators, and are not recommended for use when writing new SQL statements.

Quantified Predicate



Notes:

1 Other comparison operators are also supported [38].

A *quantified predicate* compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the predicate operator (SQLSTATE 428C4). The fullselect may return any number of rows.

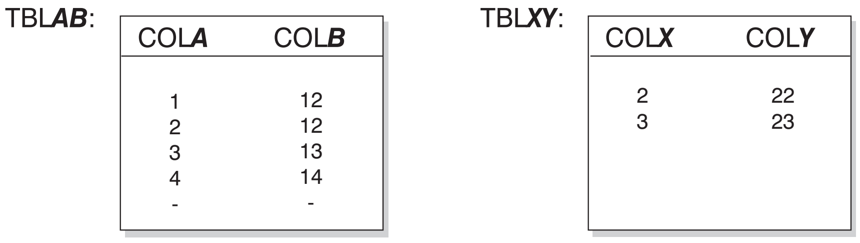
When ALL is specified:

- The result of the predicate is true if the fullselect returns no values or if the specified relationship is true for every value returned by the fullselect.
- The result is false if the specified relationship is false for at least one value returned by the fullselect.
- The result is unknown if the specified relationship is not false for any values returned by the fullselect and at least one comparison is unknown because of the null value.

When SOME or ANY is specified:

- The result of the predicate is true if the specified relationship is true for each value of at least one row returned by the fullselect.
- The result is false if the fullselect returns no rows or if the specified relationship is false for at least one value of every row returned by the fullselect.
- The result is unknown if the specified relationship is not true for any of the rows and at least one comparison is unknown because of a null value.

Examples: Use the following tables when referring to the following examples.



TBLAB:

COLA	COLB
1	12
2	12
3	13
4	14
-	-

TBLXY:

COLX	COLY
2	22
3	23

Figure 12.

Example 1

```
SELECT COLA FROM TBLAB
   WHERE COLA = ANY(SELECT COLX FROM TBLXY)
```

Results in 2,3. The subselect returns (2,3). COLA in rows 2 and 3 equals at least one of these values.

Example 2

```
SELECT COLA FROM TBLAB
   WHERE COLA > ANY(SELECT COLX FROM TBLXY)
```

Results in 3,4. The subselect returns (2,3). COLA in rows 3 and 4 is greater than at least one of these values.

Example 3

```
SELECT COLA FROM TBLAB
   WHERE COLA > ALL(SELECT COLX FROM TBLXY)
```

Results in 4. The subselect returns (2,3). COLA in row 4 is the only one that is greater than both these values.

Example 4

```
SELECT COLA FROM TBLAB
   WHERE COLA > ALL(SELECT COLX FROM TBLXY
                     WHERE COLX<0)
```

Results in 1,2,3,4, null. The subselect returns no values. Thus, the predicate is true for all rows in TBLAB.

Example 5

```
SELECT * FROM TBLAB
  WHERE (COLA,COLB+10) = SOME (SELECT COLX, COLY FROM TBLXY)
```

The subselect returns all entries from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

Example 6

```
SELECT * FROM TBLAB
   WHERE (COLA,COLB) = ANY (SELECT COLX,COLY-10 FROM TBLXY)
```

The subselect returns COLX and COLY-10 from TBLXY. The predicate is true for the subselect, hence the result is as follows:

```
COLA        COLB
----------- -----------
          2          12
          3          13
```

BETWEEN Predicate

```
►►──expression──┬─────┬──BETWEEN──expression──AND──expression────────────────►◄
                └─NOT─┘
```

The BETWEEN predicate compares a value with a range of values.

The BETWEEN predicate:

```
value1 BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
value1 >= value2 AND value1 <= value3
```

The BETWEEN predicate:

```
value1 NOT BETWEEN value2 AND value3
```

is equivalent to the search condition:

```
NOT(value1 BETWEEN value2 AND value3); that is,
value1 < value2 OR value1 > value3.
```

The values for the expressions in the BETWEEN predicate can have different code pages. The operands are converted as if the above equivalent search conditions were specified.

The first operand (expression) cannot include a function that is variant or has an external action (SQLSTATE 426804).

Given a mixture of datetime values and string representations of datetime values, all values are converted to the data type of the datetime operand.

Examples:

Example 1

```
EMPLOYEE.SALARY BETWEEN 20000 AND 40000
```

Results in all salaries between $20,000.00 and $40,000.00.

Example 2

```
SALARY NOT BETWEEN 20000 + :HV1 AND 40000
```

Assuming :HV1 is 5000, results in all salaries below $25,000.00 and above $40,000.00.

Example 3

Given the following:

Table 12.

Expressions	Type	Code Page
HV_1	host variable	437
HV_2	host variable	437
Col_1	column	850

When evaluating the predicate:

```
  :HV_1 BETWEEN :HV_2 AND COL_1
```

It will be interpreted as:

```
  :HV_1 >= :HV_2
AND :HV_1 <= COL_1
```

The first occurrence of :HV_1 will remain in the application code page since it is being compared to :HV_2 which will also remain in the application code page. The second occurrence of :HV_1 will be converted to the database code page since it is being compared to a column value.

EXISTS Predicate

```
►►──EXISTS──(fullselect)──────────────────────────────────────────────────►◄
```

The EXISTS predicate tests for the existence of certain rows.

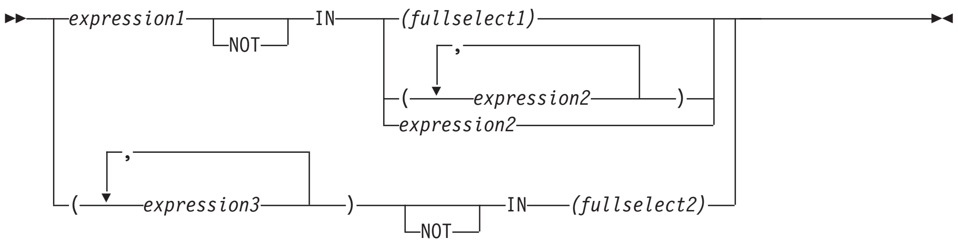
The fullselect may specify any number of columns, and

- The result is true only if the number of rows specified by the fullselect is not zero.
- The result is false only if the number of rows specified is zero
- The result cannot be unknown.

Example:

```
EXISTS (SELECT * FROM TEMPL WHERE SALARY < 10000)
```

IN Predicate



The IN predicate compares a value or values with a collection of values.

The fullselect must identify a number of columns that is the same as the number of expressions specified to the left of the IN keyword (SQLSTATE 428C4). The fullselect may return any number of rows.

- An IN predicate of the form:
  ```
  expression IN expression
  ```
  is equivalent to a basic predicate of the form:
  ```
  expression = expression
  ```
- An IN predicate of the form:
  ```
  expression IN (fullselect)
  ```
  is equivalent to a quantified predicate of the form:
  ```
  expression = ANY (fullselect)
  ```
- An IN predicate of the form:
  ```
  expression NOT IN (fullselect)
  ```
  is equivalent to a quantified predicate of the form:
  ```
  expression <> ALL (fullselect)
  ```
- An IN predicate of the form:
  ```
  expression IN (expressiona, expressionb, ..., expressionk)
  ```
  is equivalent to:
  ```
  expression = ANY (fullselect)
  ```
  where fullselect in the values-clause form is:
  ```
  VALUES (expressiona), (expressionb), ..., (expressionk)
  ```
- An IN predicate of the form:
  ```
  (expressiona, expressionb,..., expressionk) IN (fullselect)
  ```

is equivalent to a quantified predicate of the form:

```
(expressiona, expressionb,..., expressionk) = ANY (fullselect)
```

The values for *expression1* and *expression2* or the column of *fullselect1* in the IN predicate must be compatible. Each *expression3* value and its corresponding column of *fullselect2* in the IN predicate must be compatible. The "Rules for Result Data Types" on page 107 can be used to determine the attributes of the result used in the comparison.

The values for the expressions in the IN predicate (including corresponding columns of a fullselect) can have different code pages. If a conversion is necessary then the code page is determined by applying "Rules for String Conversions" on page 111 to the IN list first and then to the predicate using the derived code page for the IN list as the second operand.

Examples:

Example 1: The following evaluates to true if the value in the row under evaluation in the DEPTNO column contains D01, B01, or C01:

```
DEPTNO IN ('D01', 'B01', 'C01')
```

Example 2: The following evaluates to true only if the EMPNO (employee number) on the left side matches the EMPNO of an employee in department E11:

```
EMPNO IN (SELECT EMPNO FROM EMPLOYEE WHERE WORKDEPT = 'E11')
```

Example 3: Given the following information, this example evaluates to true if the specific value in the row of the COL_1 column matches any of the values in the list:

Table 13. IN Predicate example

Expressions	Type	Code Page
COL_1	column	850
HV_2	host variable	437
HV_3	host variable	437
CON_1	constant	850

When evaluating the predicate:

```
COL_1 IN (:HV_2, :HV_3, CON_4)
```

The two host variables will be converted to code page 850 based on the "Rules for String Conversions" on page 111.

Example 4: The following evaluates to true if the specified year in EMENDATE (the date an employee activity on a project ended) matches any of the values specified in the list (the current year or the two previous years):

```
  YEAR(EMENDATE) IN (YEAR(CURRENT DATE),
                     YEAR(CURRENT DATE - 1 YEAR),
                     YEAR(CURRENT DATE - 2 YEARS))
```

Example 5: The following evaluates to true if both ID and DEPT on the left side match MANAGER and DEPTNUMB respectively for any row of the ORG table.

```
(ID, DEPT) IN (SELECT MANAGER, DEPTNUMB FROM ORG)
```

LIKE Predicate

```
►►──match-expression──┬─────┬──LIKE──pattern-expression──────────────►
                      └─NOT─┘

►──┬──────────────────────────────┬─────────────────────────────────►◄
   └─ESCAPE──escape-expression────┘
```

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified by a string in which the underscore and percent sign may have special meanings. Trailing blanks in a pattern are part of the pattern.

If the value of any of the arguments is null, the result of the LIKE predicate is unknown.

The values for *match-expression*, *pattern-expression*, and *escape-expression* are compatible string expressions. There are slight differences in the types of string expressions supported for each of the arguments. The valid types of expressions are listed under the description of each argument.

None of the expressions can yield a distinct type. However, it can be a function that casts a distinct type to its source type.

match-expression
An expression that specifies the string that is to be examined to see if it conforms to a certain pattern of characters.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable (including a locator variable or a file reference variable)
- a scalar function
- a large object locator
- a column name
- an expression concatenating any of the above

pattern-expression
An expression that specifies the string that is to be matched.

The expression can be specified by any one of:

- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above

- an expression concatenating any of the above

with the restrictions that:

- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition it cannot be a BLOB file reference variable.
- The actual length of *pattern-expression* cannot be more than 32 672 bytes.

A **simple description** of the use of the LIKE pattern is that the pattern is used to specify the conformance criteria for values in the *match-expression* where:

- The underscore character (_) represents any single character.
- The percent sign (%) represents a string of zero or more characters.
- Any other character represents itself.

If the *pattern-expression* needs to include either the underscore or the percent character, the *escape-expression* is used to specify a character to precede either the underscore or percent character in the pattern.

A **rigorous description** of the use of the LIKE pattern follows. Note that this description ignores the use of the *escape-expression*; its use is covered later.

- Let *m* denote the value of *match-expression* and let *p* denote the value of *pattern-expression*. The string *p* is interpreted as a sequence of the minimum number of substring specifiers so each character of *p* is part of exactly one substring specifier. A substring specifier is an underscore, a percent sign, or any non-empty sequence of characters other than an underscore or a percent sign.

  The result of the predicate is unknown if *m* or *p* is the null value. Otherwise, the result is either true or false. The result is true if *m* and *p* are both empty strings or there exists a partitioning of *m* into substrings such that:

  – A substring of *m* is a sequence of zero or more contiguous characters and each character of *m* is part of exactly one substring.

  – If the *n*th substring specifier is an underscore, the *n*th substring of *m* is any single character.

  – If the *n*th substring specifier is a percent sign, the *n*th substring of *m* is any sequence of zero or more characters.

  – If the *n*th substring specifier is neither an underscore nor a percent sign, the *n*th substring of *m* is equal to that substring specifier and has the same length as that substring specifier.

  – The number of substrings of *m* is the same as the number of substring specifiers.

It follows that if *p* is an empty string and *m* is not an empty string, the result is false. Similarly, it follows that if *m* is an empty string and *p* is not an empty string, the result is false.

The predicate *m* NOT LIKE *p* is equivalent to the search condition NOT (*m* LIKE *p*).

When the *escape-expression* is specified, the *pattern-expression* must not contain the escape character identified by the *escape-expression* except when immediately followed by the escape character, the underscore character or the percent sign character (SQLSTATE 22025).

If the *match-expression* is a character string in an MBCS database then it can contain **`mixed data`**. In this case, the pattern can include both SBCS and MBCS characters. The special characters in the pattern are interpreted as follows:
- An SBCS underscore refers to one SBCS character.
- A DBCS underscore refers to one MBCS character.
- A percent (either SBCS or DBCS) refers to a string of zero or more SBCS or MBCS characters.

escape-expression

This optional argument is an expression that specifies a character to be used to modify the special meaning of the underscore (_) and percent (%) characters in the *pattern-expression*. This allows the LIKE predicate to be used to match values that contain the actual percent and underscore characters.

The expression can be specified by any one of:
- a constant
- a special register
- a host variable
- a scalar function whose operands are any of the above
- an expression concatenating any of the above

with the restrictions that:
- No element in the expression can be of type LONG VARCHAR, CLOB, LONG VARGRAPHIC or DBCLOB. In addition, it cannot be a BLOB file reference variable.
- The result of the expression must be one SBCS or DBCS character or a binary string containing exactly 1 byte (SQLSTATE 22019).

When escape characters are present in the pattern string, an underscore, percent sign, or escape character can represent a literal occurrence of itself. This is true if the character in question is preceded by an odd number of successive escape characters. It is not true otherwise.

In a pattern, a sequence of successive escape characters is treated as follows:

- Let S be such a sequence, and suppose that S is not part of a larger sequence of successive escape characters. Suppose also that S contains a total of n characters. Then the rules governing S depend on the value of n:
  - If n is odd, S must be followed by an underscore or percent sign (SQLSTATE 22025). S and the character that follows it represent (n-1)/2 literal occurrences of the escape character followed by a literal occurrence of the underscore or percent sign.
  - If n is even, S represents n/2 literal occurrences of the escape character. Unlike the case where n is odd, S could end the pattern. If it does not end the pattern, it can be followed by any character (except, of course, an escape character, which would violate the assumption that S is not part of a larger sequence of successive escape characters). If S is followed by an underscore or percent sign, that character has its special meaning.

Following is a illustration of the effect of successive occurrences of the escape character (which, in this case, is the back slash (\)).

Pattern string	Actual Pattern
\%	A percent sign
\\%	A back slash followed by zero or more arbitrary characters
\\\%	A back slash followed by a percent sign

The code page used in the comparison is based on the code page of the *match-expression* value.

- The *match-expression* value is never converted.
- If the code page of *pattern-expression* is different from the code page of *match-expression*, the value of *pattern-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).
- If the code page of *escape-expression* is different from the code page of *match-expression*, the value of *escape-expression* is converted to the code page of *match-expression*, unless either operand is defined as FOR BIT DATA (in which case there is no conversion).

Examples

- Search for the string 'SYSTEMS' appearing anywhere within the PROJNAME column in the PROJECT table.

```
SELECT PROJNAME FROM PROJECT
  WHERE PROJECT.PROJNAME LIKE '%SYSTEMS%'
```

- Search for a string with a first character of 'J' that is exactly two characters long in the FIRSTNME column of the EMPLOYEE table.

```
SELECT FIRSTNME FROM EMPLOYEE
  WHERE EMPLOYEE.FIRSTNME LIKE 'J_'
```

- Search for a string of any length, with a first character of 'J', in the FIRSTNME column of the EMPLOYEE table.

```
SELECT FIRSTNME FROM EMPLOYEE
  WHERE EMPLOYEE.FIRSTNME LIKE 'J%'
```

- In the CORP_SERVERS table, search for a string in the LA_SERVERS column that matches the value in the CURRENT SERVER special register.

```
 SELECT LA_SERVERS FROM CORP_SERVERS
   WHERE CORP_SERVERS.LA_SERVERS LIKE CURRENT SERVER
```

- Retrieve all strings that begin with the sequence of characters '%_\' in column A of the table T.

```
  SELECT A FROM T WHERE T.A LIKE
'\%\_\\%' ESCAPE '\'
```

- Use the BLOB scalar function, to obtain a one byte escape character which is compatible with the match and pattern data types (both BLOBs).

```
SELECT COLBLOB FROM TABLET
  WHERE COLBLOB LIKE :pattern_var ESCAPE BLOB(X'OE')
```

NULL Predicate

```
►►──expression──IS──┬─────┬──NULL──────────────────────────────────────────►◄
                    └─NOT─┘
```

The NULL predicate tests for null values.

The result of a NULL predicate cannot be unknown. If the value of the expression is null, the result is true. If the value is not null, the result is false. If NOT is specified, the result is reversed.

Examples:

```
PHONENO IS NULL
SALARY IS NOT NULL
```

TYPE Predicate

```
►►──expression──┬─IS──┬─────┬──OF──────────────┬────────────────────────►
                │     └─NOT─┘                  │
                │ ┌─IS─┐                       │
                └─┴────┴──┬─────┬──OF DYNAMIC TYPE──┘
                          └─NOT─┘

      ┌─,──────────────────────┐
      ▼                        │
►──(──┬──────┬──typename───────┴──)───────────────────────────────────►◄
      └─ONLY─┘
```

A *TYPE predicate* compares the type of an expression with one or more user-defined structured types.

The dynamic type of an expression involving the dereferencing of a reference type is the actual type of the referenced row from the target typed table or view. This may differ from the target type of an expression involving the reference which is called the static type of the expression.

If the value of *expression* is null, the result of the predicate is unknown. The result of the predicate is true if the dynamic type of the *expression* is a subtype of one of the structured types specified by *typename*, otherwise the result is false. If ONLY precedes any *typename* the proper subtypes of that type are not considered.

If *typename* is not qualified, it is resolved using the SQL path. Each *typename* must identify a user-defined type that is in the type hierarchy of the static type of *expression* (SQLSTATE 428DU).

The DEREF function should be used whenever the TYPE predicate has an expression involving a reference type value. The static type for this form of *expression* is the target type of the reference. See "DEREF" on page 284 for more information about the DEREF function.

The syntax IS OF and OF DYNAMIC TYPE are equivalent alternatives for the TYPE predicate. Similarly, IS NOT OF and NOT OF DYNAMIC TYPE are equivalent alternatives.

Example:

A table hierarchy exists with root table EMPLOYEE of type EMP and subtable MANAGER of type MGR. Another table, ACTIVITIES, includes a column called WHO_RESPONSIBLE that is defined as REF(EMP) SCOPE EMPLOYEE.

The following is a type predicate that evaluates to true when a row corresponding to WHO_RESPONSIBLE is a manager:

```
DEREF (WHO_RESPONSIBLE) IS OF (MGR)
```

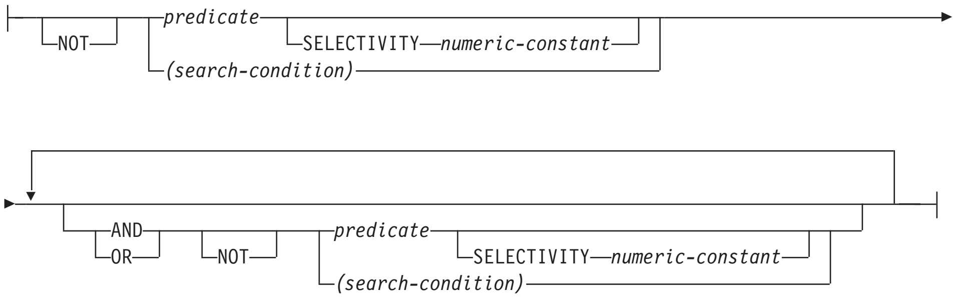
If a table contains a column EMPLOYEE of type EMP, EMPLOYEE may contain values of type EMP as well as values of its subtypes like MGR. The following predicate

```
EMPL IS OF (MGR)
```

returns true when EMPL is not null and is actually a manager.

Search Conditions

search-condition:



A *search condition* specifies a condition that is "true," "false," or "unknown" about a given row.

The result of a search condition is derived by application of the specified *logical operators* (AND, OR, NOT) to the result of each specified predicate. If logical operators are not specified, the result of the search condition is the result of the specified predicate.

AND and OR are defined in Table 14, in which P and Q are any predicates:

Table 14. Truth Tables for AND and OR

P	Q	P AND Q	P OR Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

NOT(true) is false, NOT(false) is true, and NOT(unknown) is unknown.

Search conditions within parentheses are evaluated first. If the order of evaluation is not specified by parentheses, NOT is applied before AND, and

AND is applied before OR. The order in which operators at the same precedence level are evaluated is undefined to allow for optimization of search conditions.

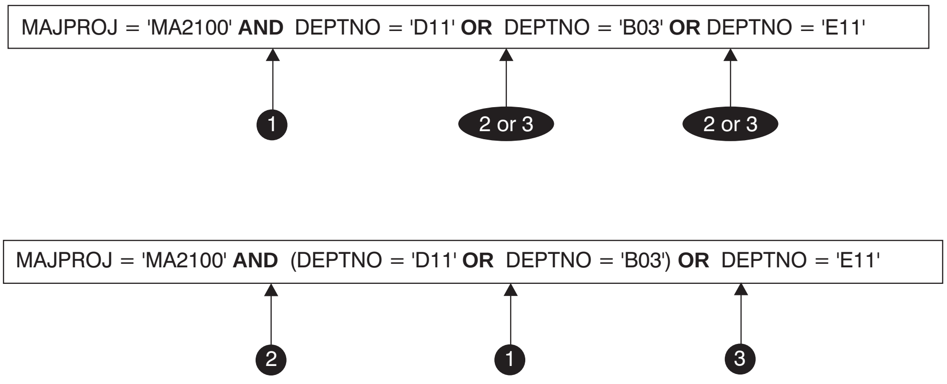

MAJPROJ = 'MA2100' AND DEPTNO = 'D11' OR DEPTNO = 'B03' OR DEPTNO = 'E11'
1
2 or 3
2 or 3
MAJPROJ = 'MA2100' AND (DEPTNO = 'D11' OR DEPTNO = 'B03') OR DEPTNO = 'E11'
2
1
3

Figure 13. Search Conditions Evaluation Order

SELECTIVITY *value*

The SELECTIVITY clause is used to indicate to DB2 what the expected selectivity percentage is for the predicate. SELECTIVITY can be specified only when the predicate is a user-defined predicate.

A user-defined predicate is a predicate that consists of a user-defined function invocation, in the context of a predicate specification that matches the predicate specification on the PREDICATES clause of CREATE FUNCTION. For example, if the function `foo` is defined with PREDICATES WHEN=1..., then the following use of SELECTIVITY is valid:

```
SELECT *
     FROM STORES
     WHERE foo(parm,parm) = 1 SELECTIVITY 0.004
```

The selectivity value must be a numeric literal value in the inclusive range from 0 to 1 (SQLSTATE 42615). If SELECTIVITY is not specified, the default value is 0.01 (that is, the user-defined predicate is expected to filter out all but one percent of all the rows in the table). The SELECTIVITY default can be changed for any given function by updating its SELECTIVITY column in the SYSSTAT.FUNCTIONS view. An error will be returned if the SELECTIVITY clause is specified for a non user-defined predicate (SQLSTATE 428E5).

A user-defined function (UDF) can be applied as a user-defined predicate and, hence, is potentially applicable for index exploitation if:

- the predicate specification is present in the CREATE FUNCTION statement
- the UDF is invoked in a WHERE clause being compared (syntactically) in the same way as specified in the predicate specification
- there is no negation (NOT operator)

Examples

In the following query, the `within` UDF specification in the WHERE clause satisfies all three conditions and is considered a user-defined predicate. (For more information about the `within` and `distance` UDFs, see the Examples section of "CREATE FUNCTION (External Scalar)" on page 590.)

```
SELECT *
   FROM customers
   WHERE within(location, :sanJose) = 1 SELECTIVITY  0.2
```

However, the presence of `within` in the following query is not index-exploitable due to negation and is not considered a user-defined predicate.

```
SELECT *
   FROM customers
   WHERE NOT(within(location, :sanJose) = 1) SELECTIVITY  0.3
```

In the next example, consider identifying customers and stores that are within a certain distance of each other. The distance of one store to another is computed by the radius of the city that the customers live in.

```
SELECT *
   FROM customers, stores
   WHERE distance(customers.loc, stores.loc) < CityRadius(stores.loc) SELECTIVITY
```

In the above query, the predicate in the WHERE clause is considered a user-defined predicate. The result produced by CityRadius is used as a search argument to the range producer function.

However, since the result produced by CityRadius is used as a range producer function, the above user-defined predicate will not be able to make use of the index extension defined on the stores.loc column. Therefore, the UDF will make use of only the index defined on the customers.loc column.