Contents

Preface ........................................ vi
Topics covered in the Reference Manual .......................... vi
Product Documentation Set ................................. vii

Chapter 1

SQL Support in Composite .......................... 1
Data Types ........................................... 2
  BINARY ........................................... 2
  BLOB and CLOB ................................... 2
  DECIMAL ........................................ 2
  INTEGER ........................................ 2
  INTERVAL DAY and INTERVAL YEAR ...................... 2
  String ......................................... 5
  XML .......................................... 5
Functions ............................................. 6
  Aggregate Functions ................................ 7
    Character Functions ............................... 16
    Conditional Function ............................ 33
    Convert Functions ................................ 39
    Date Functions ................................ 54
    Numeric Functions ................................ 59
    XML Functions ................................ 69
Operators .......................................... 80
  Arithmetic Operators ................................ 80
  Comparison Operators ............................... 97
  Logical Operators ................................ 102
  Condition Operators ................................ 103
SQL Keywords ...................................... 114
Using SQL Keywords ................................ 114
Subqueries ......................................... 138
  Scalar Subqueries and Correlated Subqueries .......... 139
Consolidated List of Reserved Words ....................... 140
Chapter 3

Composite Query Engine Options .................................. 229
SELECT Options .................................................. 230
case_sensitive ................................................. 230
ignore_trailing_spaces ........................................ 231
disable_data_cache .......................................... 231
disable_statistics ............................................. 231
disable_cbo .................................................... 232
max_rows_limit ............................................... 232
force_disk ...................................................... 232
disable_threads .............................................. 233
disable_plan_cache ......................................... 233
disable_push .................................................. 233
strict .......................................................... 234
union / intersect / except Options ...................... 234
parallel ......................................................... 234
force_disk ...................................................... 235
disable_push .................................................. 235
left_cardinality ............................................... 238
right_cardinality ............................................. 238
force_disk ...................................................... 239
disable_threads .............................................. 239
disable_push .................................................. 239
partition_size ................................................ 240
group by Options ............................................ 241
force_disk ...................................................... 241
disable_threads .............................................. 241
disable_push .................................................. 241
order by Options ............................................. 242
force_disk ...................................................... 242
disable_threads .............................................. 242
disable_push .................................................. 242
insert / update / delete Options ...................... 244
case_sensitive ................................................. 244
while .......................................................... 221
examples ....................................................... 222
Preface

This reference manual contains information that you can refer to when developing client applications for Composite.

This manual is for anyone with a knowledge of relational data sources, hierarchical data sources, and data modeling.

For information on product documentation, see “Product Documentation Set” on page vii.

Topics covered in the Reference Manual

- SQL Support in Composite
  Describes with examples the SQL functions and reserved words supported in Composite.

- Composite SQL Script
  Provides complete reference to Composite’s own SQL Script language. It does not provide advanced-level programming tutorials.

- Composite Built-in Procedures
  Describes all the built-in procedures in the procedure library.

- Data Type Mappings
  Maps Composite JDBC data types to native data types used in other data sources.

- JAVA APIs for Custom Procedures
  Provides complete reference to Composite’s Java APIs for writing custom procedures.

- Composite System Tables
  Describes the system tables in the current version of Composite.
Product Documentation Set

All the documents are available in PDF in the *docs* directory at the root level of the product installation directory, as in the following example for Windows:

`\<installation_directory>\docs\`

The following table lists the documents available for *Composite Information Server* (CIS):

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation Guide</td>
<td>Provides instructions for installing Composite on Windows and UNIX systems</td>
<td>• In PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>Administration Guide</td>
<td>Provides instructions for administering Composite on Windows and UNIX systems</td>
<td>• In PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>Getting Started with Composite</td>
<td>Introduces the Composite software and provides quick steps for obtaining a unified view solution</td>
<td>• In PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>User’s Guide</td>
<td>Describes how to benefit from Composite’s capabilities, mainly using the Studio</td>
<td>• In PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>Reference Manual</td>
<td>Contains reference information on the software technologies used in Composite. It is not a programming manual.</td>
<td>• In PDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>Help Topics</td>
<td>Guide you through the Studio UI and data modeling tasks</td>
<td>• On-line in the Studio</td>
</tr>
<tr>
<td>Release Notes</td>
<td>Provide updated information about the software</td>
<td>Contact: <a href="mailto:support@compositerw.com">support@compositerw.com</a></td>
</tr>
</tbody>
</table>

Contact: support@compositerw.com
Chapter 1

SQL Support in Composite

Composite allows query specification and data updates using standard SQL. Composite supports a strict subset of SQL2 (or, ANSI-92 SQL).

In the following sections, this chapter describes the SQL functions, operators, and keywords that are supported in Composite:

- “Data Types” on page 2
- “Functions” on page 6
  - “Aggregate Functions” on page 7
  - “Character Functions” on page 16
  - “Conditional Function” on page 33
  - “Convert Functions” on page 39
  - “Date Functions” on page 54
  - “Numeric Functions” on page 59
  - “XML Functions” on page 69
- “Operators” on page 80
  - “Arithmetic Operators” on page 80
  - “Comparison Operators” on page 97
  - “Logical Operators” on page 102
  - “Condition Operators” on page 103
- “SQL Keywords” on page 114
  - “Using SQL Keywords” on page 114
- “Subqueries” on page 138
- “Consolidated List of Reserved Words” on page 140
Data Types

This section provides special notes on the following data types supported in Composite:

* BINARY, BLOB and CLOB, DECIMAL, INTEGER, INTERVAL DAY and INTERVAL YEAR, String, and XML

**BINARY**

*BINARY* type (BINARY, VARBINARY) behaves similar to the *String* type, padding zero bytes instead of spaces.

**BLOB and CLOB**

*BLOB* and *CLOB*

- You can project (that is, select) BLOB/CLOB columns.
- Currently, you can use BLOB or CLOB only in the *CAST* function.

**DECIMAL**

*DECIMAL* type (DECIMAL, NUMERIC)

- Throws an error if the number of digits to the left of the decimal does not fit in the number. For example, "12345.00" in DECIMAL(4,2).
- Rounds or extends the precision of the part to the right of the decimal.
  - "1.25" -> "1.3" or "1.25" -> "1.250"

**INTEGER**

*INTEGER* type (TINYINT, SMALLINT, INTEGER, BIGINT) throws a runtime error if the value is out of the valid range for the integer.

**INTERVAL DAY and INTERVAL YEAR**

Two data types—INTERVAL DAY and INTERVAL YEAR—can be used in arithmetic operations (addition, subtraction, division, and multiplication), and functions such as ABS, CAST, and EXTRACT.
The `INTERVAL DAY` and `INTERVAL YEAR` data types are distinct and are not directly compatible with one another. They both represent a duration of time, with different units of measurement.

Intervals may be negative.

**INTERVAL DAY**

`INTERVAL DAY` represents a duration of time that can be measured in days, hours, minutes, and (nano) seconds. The unit may be expressed as days only, hours only, minutes only, seconds only, both days and hours, days to minutes, hours to seconds, and so on. All interval day expressions are compatible with other `INTERVAL DAY` expressions.

**Syntax - INTERVAL DAY**

The interval day literal has the following syntax:

- `INTERVAL 'dd hh:mm:ss.nn' DAY TO SECOND`
- `INTERVAL 'dd hh:mm' DAY TO MINUTE`
- `INTERVAL 'dd hh' DAY TO HOUR`
- `INTERVAL 'dd' DAY`
- `INTERVAL 'hh' HOUR`
- `INTERVAL 'mm' MINUTE`
- `INTERVAL 'ss.nn' SECOND`

A space separates the day value from the hour value. A colon separates the hours, minutes, and second values from each other. A decimal place separates the fractional seconds from the seconds.

If no precision is given, a default leading precision of 2 is specified.

- `INTERVAL '3' DAY`
- `INTERVAL '3' DAY(2)`

The two expressions above are equivalent. The maximum leading precision is 9.

A default fractional precision of 6 is specified if seconds are involved. The fractional precision limits the number of decimal places to the right of the decimal place. Unlike the leading precision, digits will be automatically truncated if the fractional precision is exceeded.

- `INTERVAL '3' minute(3) to seconds(6)`
The two expressions above are equivalent. If second is the unit of measurement, then the precision is specified with the syntax:

\[
\text{INTERVAL '3.99' second(2, 6)}
\]

A comma separates the leading precision from the fractional precision. 9 digits is the maximum fractional precision. 0 is a valid fractional precision.

\[
\text{INTERVAL '9:59' minutes to second(0)}
\]

The expression above will truncate any fractional seconds.

**INTERVAL YEAR**

INTERVAL YEAR represents a unit of time that is measured in months and years. It may be expressed in years only, months only, or both year and months.

INTERVAL YEAR is not compatible with INTERVAL DAY because a year may consist of either 365 days or 366 days in a leap year. A month can consist of either 28, 29, 30 or 31 days.

**Syntax - INTERVAL YEAR**

\[
\text{INTERVAL '12' YEAR [TO MONTH]} \\
\text{INTERVAL '3' MONTH}
\]

If year-month is the unit of measurement, the syntax is

\[
\text{INTERVAL '2-11' YEAR TO MONTH}
\]

A dash separates the year and month values. The month value must not exceed 11 months. Negative intervals may be represented in three ways:

\[
-\text{INTERVAL '3' MONTH} \\
\text{INTERVAL '-3' MONTH} \\
\text{INTERVAL -'3' MONTH}
\]

In fact, all three may be used at once

\[
-\text{INTERVAL -'-3' MONTH}
\]

which results in a -3 month interval.

A default precision of 2 is assigned if none is specified. For example,

\[
\text{INTERVAL '99' year(2)}
\]
INTERVAL '99' year
Both expressions above are the same. The precision is a number indicating the maximum number of digits in the leading number. For example, the expression below is invalid because the precision is exceeded by the five digits in the year value.
INTERVAL '20001' year(4)
If it is a year-month interval, the precision only applies to the year
INTERVAL '200-09' year(3) to month
The month is limited to values below 12 in a year-month interval. The maximum number of years is 999,999,999. Therefore the precision is limited to 9 digits.
→ For other details on using INTERVAL DAY and INTERVAL YEAR in arithmetic operations and functions, see the corresponding sections: Add (+), Divide (÷), Multiply (×), Negate (−), Subtract (−), ABS, CAST, EXTRACT

String
String type (CHAR, VARCHAR)
- If less than min length (only happens with CHAR), spaces are padded to fill out the string.
- The CHAR type is now padded while it was not generally padded before.
“CONCAT(char10, char10)” may result in “A    B” instead of “AB”.

XML
Composite’s support for the XML data type complies with the ANSI 9075 section 14 XML specification.

Syntax
XML [ ( { DOCUMENT | CONTENT | SEQUENCE } ]
[ ( { ANY | UNTYPED | XMLSCHEMA schema-details } ]
) ]
}
Functions

Function is a pre-defined, named routine that performs a special task. Operator is an arithmetic symbol that performs a special task. Keyword is a word in SQL that is reserved as a part of syntax. Depending on the SQL statement, a keyword can be either a mandatory element of the statement or optional. Keyword is also known as reserved word.

Functions, operators, and keywords have a special significance in SQL and therefore cannot be used for naming a Composite resource.

When you design a query in the Model panel of the view editor in the Modeler, the SQL of the query is automatically generated and displayed in the SQL panel, which you access
Functions

through the SQL tab in the editor. You can also use the SQL panel in the Modeler to type your SQL statements.

To insert the functions and operators from the Grid panel, right-click over a Column or Criteria cell in the Grid panel of the view editor in the Studio, and select Function. See To include a function in the SQL via the Grid panel in the chapter Views in the User’s Guide to know how to include functions in your SQL.

Composite supports the following types of functions:

- **AGGREGATE** (page 7), **CHARACTER** (page 16), **CONDITIONAL** (page 33), **CONVERT** (page 39), **DATE** (page 54), **NUMERIC** (page 59), and **XML** (page 69).

These functions accept zero, one, or more arguments. Zero argument means no argument. An argument can be a column name, literal value, or another function. This section describes the functions supported in Composite with examples.

### Aggregate Functions

Composite supports the following aggregate functions:

- **AVG**, **COUNT**, **MAX**, **MIN**, **SUM**, and **XMLAGG**

Each of these functions takes one argument of a specific type and returns an output of a specific type.

See also “DISTINCT in Aggregate Functions” on page 14.

**Note**

If any column in the SELECT clause is outside of an aggregate function, you must include the column also in the GROUP BY clause. See the example given for **AVG**.

### AVG

Given a set of numeric values, this function calculates and returns the average of the input values, as **FLOAT**, **DECIMAL**, or **NULL**. See Table 2, “Input and output types for **AVG**” for input and output types.

**Syntax**

```
AVG(expression)
```

where `expression` is a numeric expression.
Remarks
- AVG works only with numeric data types.
- If you want to exclude a specific row from the calculation of the average, make any column value in the row NULL.
- If the input is a set of empty rows, the output is NULL.

Example
SELECT AVG(UnitPrice) Price, ProductID
FROM /shared/examples/ds_inventory/products products
GROUP BY ProductID

The following table lists the input types that you can use in AVG and the corresponding output type you will receive.

Table 2. Input and output types for AVG

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL (p, s)</td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td></td>
<td>NUMERIC (p, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
COUNT

Counts the number of rows in a specified table/column.

**Syntax 1**

```
COUNT(expression)
```

where `expression` is a column.

**Syntax 2**

```
COUNT(*)
```

**Remarks**

- The values in the specified column can be of any data type.
- `COUNT(*)` returns the count of all rows, including the NULL rows.
- If the input is a non-NULL set of values, the output is a positive integer.
- If the input is NULL, the output is 0 (zero).

**Example**

```
SELECT
    COUNT(products.ProductID) CountColumn
FROM
    /shared/examples/4s_inventory/products products
```

The following table lists the input types that you can use in `COUNT`, and the corresponding output type you will receive.
Functions

Chapter 1  SQL Support in Composite

MAX

Given a set of values, this function returns the maximum value in the input set.

Syntax

\[
\text{MAX}(\text{expression})
\]

where \text{expression} can be numeric, string, or date-time.

Remarks

- The output type is the same as that of the input.
- If the input is a \texttt{CHAR}, the output is the highest string in the sorted order.

---

Table 3. Input and output types for COUNT

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>BINARY</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>VARBINARY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLOB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CLOB</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL Zero</td>
<td></td>
</tr>
</tbody>
</table>
If the input is date/time, the output is the latest date/time.
If the input is a literal, the output is the same literal.
If the input is a numeric expression, the function `MAX` compares the values in algebraic order. That is, large negative numbers are less than small negative numbers, which are less than zero.

**Example**

```sql
SELECT
    MAX(products.UnitPrice) Price,
    MAX(orders.OrderDate) Date
FROM
    /shared/examples/da_inventory/products products,
    /shared/examples/da_orders/orders orders
```

The following table lists the input types that you can use in `MAX`, and the corresponding output type you will receive.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>CHAR</td>
<td>Same type as that of the input.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td>For example, if the input is of type CHAR, the output type is also CHAR.</td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
MIN
Given a set of values, this function returns the minimum value in the input set.

Syntax
\[ \text{MIN}(\text{expression}) \]
where \( \text{expression} \) can be numeric, string, or date/time.

Remarks
- The output type is the same as that of the input.
- If the input is a \texttt{CHAR}, the output is the lowest string in the sorted order.
- If the input is date/time, the output is the earliest date/time.
- If the input is a literal, the output is the same literal.
- If the input is a numeric expression, the function \texttt{MIN} compares the values in algebraic order. That is, large negative numbers are less than small negative numbers, which are less than zero.

Example
\[
\begin{align*}
\text{SELECT} & & \\
\text{MIN}(\text{products.UnitPrice}) & \text{Expr1}, & \\
\text{MIN}(\text{orders.OrderDate}) & \text{Expr2} & \\
\text{FROM} & & \\
/\text{shared/examples/ds_inventory/products} & \text{products}, & \\
/\text{shared/examples/ds_orders/orders} & \text{orders} & \\
\end{align*}
\]
The following table lists the input types that you can use in MIN, and the corresponding output type you will receive.

**Table 5. Input and output types for MIN**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>CHAR, VARCHAR, LONGVARCHAR, DATE, TIME, TIMESTAMP, TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC, FLOAT, REAL, DECIMAL, NULL</td>
<td>Same type as that of the input. For example, if the input is of type CHAR, the output type will also be CHAR.</td>
</tr>
</tbody>
</table>

**SUM**

Given a set of numeric values, this function returns the sum-total of all the values in the input set.

**Syntax**

```
SUM(expression)
```

where `expression` is a numeric expression.

**Remarks**

- Works only with numeric data types.
- The sum of an empty table (table with no rows) cannot be evaluated.
- If the input is a set of empty rows, the output is NULL.
Example

```sql
SELECT SUM(products.UnitPrice) Total
FROM /shared/examples ds_inventory/product products
```

The following table lists the input types that you can use in `SUM`, and the corresponding output type you will receive.

**Table 6. Input and output types for `SUM`**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL DECIMAL(p,s) NUMERIC(p,s)</td>
<td>FLOAT DECIMAL(p+s,s)</td>
</tr>
<tr>
<td>SUM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**DISTINCT in Aggregate Functions**

By default, aggregate functions operate on all the values supplied. You can use the `DISTINCT` keyword to eliminate duplicate values in aggregate function calculations.

`DISTINCT` in the `SELECT` clause and `DISTINCT` in an aggregate function do not return the same result.

Including a non-`DISTINCT` aggregate function and a `DISTINCT` aggregate function in the same `SELECT` clause can produce misleading results. Either all of the aggregate functions or none should be used with `DISTINCT` in the `SELECT` clause.
**Syntax**

aggregate-function([ALL | DISTINCT] expression)

**Example**

```
SELECT COUNT(DISTINCT customer_id) FROM orders
```

**XMLAGG**

The XML aggregate function XMLAGG works on columns. This function is valid where other aggregate functions are valid.

This function accepts one argument which will be aggregated across the groups specified in the **GROUP BY** clause if that clause is specified.

**Syntax**

```
XMLAGG [ ORDER BY <sort specification list> ]
```

**Remarks**

- The aggregation can be ordered with an **ORDER BY** clause specific to the XML aggregate function. This is independent of the **SELECT** **ORDER BY** clause.
- If the argument evaluates to **NULL**, the result will be **NULL**.

**Example of XMLAGG without ORDER BY**

```
SELECT CAST(XMLAGG(XMLELEMENT(name Name, ContactLastName))
    AS VARCHAR(10000)) 'Last Name'
FROM /shared/examples/ds_orders/customers CUSTOMER
WHERE CustomerID < 23
```
Example of XMLAGG with ORDER BY

```sql
SELECT XMLAGG((XMLELEMENT(name Details, XMLATTRIBUTES(ProductID as product), XMLELEMENT(name orderno, OrderID), XMLELEMENT(name status, Status), XMLELEMENT(name price, UnitPrice))))
ORDER BY ProductID ASC, Status ASC, OrderID DESC, UnitPrice ASC
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

Character Functions

Composite supports the following CHARACTER functions:

- CHAR_LENGTH (see LENGTH), CHR, CONCAT, LENGTH, LOWER, POSITION, REPLACE, RTRIM, SPACE, SUBSTRING, TRIM, and UPPER

Of these functions, LENGTH, LOWER, RTRIM, SPACE, TRIM, and UPPER, take one argument of a specific type and returns an output of a specific type. CONCAT takes two arguments and combine them, whereas REPLACE, and SUBSTRING take three arguments.

CHR

CHR is a string function that converts an integer ASCII code to a character.

**Syntax**

```sql
CHR(integer)
```

**Remarks**

- CHR can accept a String input as long as the String can be converted to a numeric value.
- The input must be equal to 0 (zero) or a value that is between 0 and 255.
If the input is **NULL**, the output will be **NULL**.

- If the input is less than zero, an exception will be thrown.
- If the input is greater than the maximum value of `INTEGER` (2147483647), an exception will be thrown.
- For an ASCII chart, see: [http://www.techonthenet.com/ascii/chart.php](http://www.techonthenet.com/ascii/chart.php)

**Example**

```sql
SELECT DISTINCT CHR(100) FROM /shared/examples/ds_orders/customers
```

Table 7. Input and output types for CHR

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR</td>
<td>TINYINT</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>CHR</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**CONCAT**

Given two arguments, the `CONCAT` function concatenates the second argument to the first.

**CONCAT Operator** (`||`)

Given two expressions, this operator concatenates the second one to the first.

**Syntax**

`CONCAT(string1, string2)`

The arguments of `CONCAT` can be of string type or non-string type, and you can concatenate them in any combination. For example, `CONCAT(<string>, <string>)` or
CONCAT(<string>, <non-string>), CONCAT(<non-string>, <string>), or
CONCAT(<non-string>, <non-string>).

Remarks

- You can concatenate as many strings (or non-strings) as you want in order to
  create one single string out of all the concatenated strings.
- To concatenate a non-string to a string, use the `CAST` function to convert the
  non-string to string.
- Enclose a literal string within single quotes in order to concatenate it with
  another string (or non-string). For example, `CONCAT('string1', string2)`, where `string1` is literal.
- The `CONCAT` function does not supply a white-space between the concatenated
  strings (or non-strings); you must provide the white-space character manually.
  Use the Subfunction button in the Function Arguments Input dialog to provide a space
  between concatenated strings, or use the format
  `CONCAT('string1', CONCAT(' ', 'string2'))`
- If any of the input strings in a `CONCAT` function is `NULL`, the result string is also
  `NULL`. Otherwise, the output type is `STRING`.

Example

```sql
SELECT
    CONCAT(customers.ContactFirstName,
        CONCAT(' ', customers.ContactLastName)) Expr1,
    CONCAT('a', CONCAT(' ', 'b')) Expr2,
    CONCAT('a', CONCAT(' ', NULL)) Expr3,
    CONCAT(NULL, CONCAT(' ', NULL)) Expr4,
    CONCAT('NULL', CONCAT(' ', NULL)) Expr5,
    CONCAT('Feb', CONCAT(' ', CAST(2004 AS BIT))) Expr6,
    CONCAT('0100' || '1010') Expr10,
```
The following table lists the input types that you can use in CONCAT, and the corresponding output type you will receive.

**Table 8. Input and output types for CONCAT**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING, BOOLEAN, DATE, TIME, TIMESTAMP</td>
<td>Any type listed in the preceding column for input argument 1.</td>
<td>STRING</td>
</tr>
</tbody>
</table>
Table 8. Input and output types for CONCAT

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT</td>
<td>NULL</td>
<td>CHAR VARCHAR LONGVARCHAR STRING BOOLEAN DATE TIME TIMESTAMP TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL</td>
<td>STRING</td>
</tr>
<tr>
<td>CONCAT</td>
<td>CHAR VARCHAR LONGVARCHAR STRING BOOLEAN DATE TIME TIMESTAMP TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
LENGTH

Returns the number of characters, not the number of bytes, in a given string expression.

Syntax

```
LENGTH(string)
```

Remarks

- `CHAR_LENGTH` and `CHARACTER_LENGTH` are synonymous with `LENGTH`.
- If the input is `NULL`, the output is also `NULL`. Otherwise, the output is an integer that is equal to or greater than zero.
- If the input is an empty string, the output is 0 (zero).
- The length of a white-space in an input argument is counted as 1 (one).
- If you want to count the white-space included in an input string, use the `CONCAT` function to accommodate the space, as in the example:
  ```sql
  LENGTH(CONCAT(customers.ContactFirstName, CONCAT(' ',
               customers.ContactLastName)))
  ```
- If you want to find the length of an integer, you must convert the integer to `VARCHAR`, and pass the string as the input for the `LENGTH` function. Suppose you want to find out the number of digits in a phone number, cast the phone number's integer into a `VARCHAR` and use it in the `LENGTH` function.

Example

```
SELECT
    LENGTH(customers.PostalCode) Expr1,
    LENGTH(NULL) Expr2,
    LENGTH(' ') Expr3,
    LENGTH('') Expr4,
    LENGTH(CONCAT(customers.ContactFirstName,
                   CONCAT(' ', customers.ContactLastName))) Expr5,
    LENGTH(customers.FaxNumber) Expr6,
    LENGTH(to_char(1000)) Expr7,
```
Functions Chapter 1 SQL Support in Composite

```
LENGTH(CAST(customers.PhoneNumber AS VARCHAR)) Expr8
FROM
/shared/examples/ds_orders/customers customers
```

The following table lists the input types that you can use in `LENGTH`, and the corresponding output type you will receive.

### Table 9. Input and output types for `LENGTH`

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LENGTH</code></td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**LOWER**

This function renders all the alphabetical characters in a given string in lower case. It is typically used to format the output and also make case-insensitive comparisons.

**Syntax**

```
LOWER(string)
```

**Remarks**

- The input string must be enclosed within single quotes.
- Non-alpha characters, such as numerals, white-spaces, and punctuations in an input string enclosed within single quotes are unaffected by the `LOWER` function.
- If the input is `NULL`, the output is also `NULL`.
- If the input is an empty string (enclosed in single quotes), the output is also an empty string.
- If the input is a white-space (enclosed in single quotes), the output is also empty.
Example of LOWER with a comparison

```sql
SELECT ContactLastName AS Name
FROM /shared/examples/ds_orders/customers
WHERE LOWER (ContactLastName) LIKE 'Ho%';
```

This example would convert all the letters in a `ContactLastName` into lower case and pull out all the names from the table `customers` containing the sequence “ho” as follows:

Howard
Honner
Nicholson
Thompson

Example of LOWER in other contexts

```sql
SELECT LOWER(products.ProductName) Name,
      LOWER('YOU') Expr4,
      LOWER(' ') Expr6,
      LOWER('YoU 9 fEEt') Expr2,
      LOWER(NULL) Expr1
FROM /shared/examples/ds_inventory/products products
```
The following table lists the input types that you can use in LOWER, and the corresponding output type you will receive.

### Table 10. Input and output types for LOWER

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER</td>
<td>CHAR</td>
<td>Same type as that of the input.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td>For example, if the input is of type CHAR, the output type will also be CHAR.</td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**POSITION**

Given two input strings, this function returns an integer value representing the starting position of the first string within the second string.

- This function is case-sensitive.
- All string types, all numeric types, and all data types are accepted as input arguments.
- The output is always an integer provided none of the input strings is NULL. Otherwise, NULL is returned.
- If any of the arguments is NULL, the function returns NULL.
- If the first argument is a blank string, the function returns 1 (one).
- If the first argument is not found within the second argument, the function returns 0 (zero).

**Examples**

- `POSITION('is' IN 'mistake')`
  Output: 2
- `POSITION(' ', IN 'mistake')`
  Output: 1
- `POSITION('no' IN 'yes')`
  Output: 0
REPLACE

Given three strings (representing the search string, string to be replaced, and replacement string respectively), this function replaces all the instances of the string to be replaced that are contained in the search string with the replacement string.

**Syntax**

```sql
REPLACE(search_string, string_to_be_replaced, replacement_string)
```

**Remarks**

- The `string_to_be_replaced` and the `replacement_string` must be of the same type (string or binary).
- All occurrences of the `string_to_be_replaced` within the `search_string` are replaced with the `replacement_string`.
- The `string_to_be_replaced` and the `replacement_string` must be enclosed within single quotes.
- If any of the input strings is `NULL`, the output is also `NULL`. Otherwise, the output is a string.

**Example**

```sql
SELECT 
  REPLACE(products.ProductName, 'USB 2.0', 'USB 3.0') Replaced
FROM 
/shared/examples/ds_inventory/products products
```

The following table lists the input types that you can use in REPLACE, and the corresponding output type you will receive.
**RTRIM**

This function trims all the white-spaces from the right-side of a string.

**Syntax**

\[
\text{RTRIM}(\text{string})
\]

**Remarks**

- White-spaces within (that is, in the middle of) an input string are not affected.
- If the input string is \text{NULL}, the output is also \text{NULL}. Otherwise, the output is of the same type as the input.

---

Table 11. Input and output types for REPLACE

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type (string to be replaced)</th>
<th>Input Argument 2 Type (search string)</th>
<th>Input Argument 3 Type (replacement string)</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>Same as that of argument 1.</td>
<td>Same as that of argument 1.</td>
<td>Same as that of argument 1.</td>
</tr>
<tr>
<td>NULL</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>Same as that of argument 2.</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>NULL</td>
<td>Same as that of argument 1.</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>Same as that of argument 1.</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

---

26
**Example**

\[
\text{concat} \left( \text{RTRIM} \left( 'AAA ' \right), \ 'Member' \right)
\]

with three white-spaces at the end of the sequence `AAA` and no white-space preceding the `M` in `Member` will produce the following result:

`AAAMember`

Whereas

\[
\text{concat} \left( \text{RTRIM} \left( 'AAA ' \right), \ 'Member' \right)
\]

with three white-spaces at the end of the sequence `AAA` and one white-space preceding the `M` in `Member` will produce the following result:

`AAA Member`

The following table lists the input types that you can use in `RTRIM`, and the corresponding output type you will receive.

**Table 12. Input and output types for RTRIM**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTRIM</td>
<td>CHAR</td>
<td>Same type as that of the input.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td>For example, if the input is of type CHAR, the output type will also be CHAR.</td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**SPACE**

This function returns a string of spaces repeated as many times as the integer specified.

**Syntax**

\[
\text{SPACE} (\text{integer})
\]

**Remarks**

- This function accepts a DECIMAL input.
- If the input is NULL, the output is also NULL. Otherwise, the output is a string.
If the input is a negative integer, the output will be NULL.

**Example**

```sql
SELECT
    CONCAT(customers.ContactFirstName,
    CONCAT(SPACE(1), customers.ContactLastName)) Name
FROM
    /shared/examples/ds_orders/customers customers
```

The following table lists the input types that you can use in SPACE, and the corresponding output type you will receive.

**Table 13. Input and output types for SPACE**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPACE</td>
<td>TINYINT</td>
<td>CHAR</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**SUBSTRING**

Given a string, this function returns the substring starting from the start position, and extending up to the length specified by the substring length.

**Syntax**

```sql
SUBSTRING(string, start_position, length_of_substring)
```

where `start_position` and `length_of_substring` are positive integers.

**Remarks**

- The original string is assumed to start at position one (1).
- The resulting substring is any sequence of characters in the original string, including an empty string.
If the original string is an empty string, the resulting substring is also an empty string.

If any of the input arguments is NULL, the output is also NULL.

**Example**

```sql
SELECT
    SUBSTRING(customers.PhoneNumber, 1, 5) AreaCode
FROM
    /shared/examples/ds_orders/customers customers
```

The following table lists the input types that you can use in `SUBSTRING` and the corresponding output type you will receive.
Table 14. Input and output types for SUBSTRING

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type (S)</th>
<th>Input Argument 2 Type (P1)</th>
<th>Input Argument 3 Type (P2)</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSTRING</td>
<td>CHAR</td>
<td>TINYINT</td>
<td>Same as that of argument 2.</td>
<td>Same as that of argument 1.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>TINYINT</td>
<td>Same as that of argument 2.</td>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>NULL</td>
<td>TINYINT</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>NULL</td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td>NULL</td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td>TINYINT</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>SMALLINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRING</td>
<td>BIGINT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TRIM

This function removes all the leading and trailing blanks in the input string.

Syntax

TRIM(string)

Remarks

- If the input string is NULL, the output is also NULL. Otherwise, the output is a string.
- If you want to trim characters within a string, use the REPLACE function, as in the example given here

Example

```sql
SELECT customers.PhoneNumber,
TRIM(customers.PhoneNumber, '('415')', '') AS "Area-code Trimmed Phone Number"
FROM
/shared/examples/ds_orders/customers customers
```

Table 15. Input and output types for TRIM

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIM</td>
<td>CHAR</td>
<td>Same as that of the input.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
UPPER

This function renders all the alphabetical characters in a given string in upper case. It is used to format the output and also make case-insensitive comparisons.

Syntax

UPPER(string)

Remarks

- The input string must be enclosed within single quotes.
- Non-alpha characters, such as numerals, white-spaces, and punctuations in an input string enclosed within single quotes are unaffected by the UPPER function.
- If the input is NULL, the output is also NULL.
- If the input is an empty string (enclosed in single quotes), the output is also an empty string.
- If the input is a white-space (enclosed in single quotes), the output is also empty.

Example

SELECT UPPER(products.ProductName) ProductName
FROM /shared/examples/ds_inventory/products products

The following table lists the input types that you can use in UPPER, and the corresponding output type you will receive.

Table 16. Input and output types for UPPER

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPER</td>
<td>CHAR</td>
<td>Same as that of the input.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Function

The conditional function supported in Composite: NULLIF

NULLIF

The NULLIF function compares two arguments and returns NULL if they are equal, or the first argument otherwise.

The first argument in NULLIF cannot be NULL. The output data type of NULLIF is always the same as the first argument. See Table 17, “Input and output types for NULLIF” on page 34.

Syntax

NULLIF(argument1, argument2)

is equivalent to

CASE
  WHEN expression1 = expression2 THEN NULL
  ELSE expression1
END

Example

SELECT ProductID, UnitPrice, NULLIF(UnitPrice, 0) as "Null Price"
FROM /shared/examples/ds_orders/products products

The following table lists the data types of the input and output arguments for NULLIF.
### Table 17. Input and output types for NULLIF

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLIF</td>
<td>CHAR</td>
<td>CHAR, VARCHAR, LONGVARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DECIMAL, NUMERIC, DATE, TIME, TIMESTAMP, BINARY, VARBINARY, NULL</td>
<td>Same as argument 1</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>CHAR</td>
<td>CHAR, VARCHAR, LONGVARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DECIMAL, NUMERIC, DATE, TIME, TIMESTAMP, BINARY, VARBINARY, NULL</td>
<td></td>
</tr>
</tbody>
</table>


Table 17. Input and output types for NULLIF

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLIF</td>
<td>VARCHAR</td>
<td>NULL</td>
<td>Same as argument 1</td>
</tr>
<tr>
<td></td>
<td>CHAR</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGBINARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BINARY</td>
<td>VARBINARY</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

| TINYINT| CHAR                  | NULL                 |                           |
|        | VARCHAR               |                      |                           |
|        | LONGBINARY           |                      |                           |
|        | TINYINT               | SMALLINT             |                           |
|        | INTEGER               | BIGINT               |                           |
|        | REAL                  | DECIMAL              |                           |
|        | DECIMAL               | NUMERIC              |                           |
|        | DATE                  | TIME                 |                           |
|        | TIME                  | TIMESTAMP            |                           |
|        | BINARY                | VARBINARY            |                           |
|        | NULL                  | NULL                 |                           |
### Table 17. Input and output types for NULLIF

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLIF</td>
<td>SMALLINT</td>
<td>CHAR, VARCHAR,</td>
<td>Same as argument 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TINYINT, SMALLINT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER, BIGINT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLOAT, REAL, DECIMAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUMERIC, TIMESTAMP,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td>CHAR, VARCHAR,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TINYINT, SMALLINT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER, BIGINT,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLOAT, REAL, DECIMAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUMERIC, TIMESTAMP,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
Table 17. Input and output types for NULLIF

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULLIF</td>
<td>BIGINT</td>
<td>CHAR</td>
<td>Same as argument 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TINYINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>CHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Input Argument 1 Type</td>
<td>Input Argument 2 Type</td>
<td>Output Type</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>NULLIF</td>
<td>REAL</td>
<td>CHAR, VARCHAR, LONGVARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DECIMAL, NUMERIC, NULL</td>
<td>Same as argument 1</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>CHAR, VARCHAR, LONGVARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DECIMAL, NUMERIC, NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>CHAR, VARCHAR, LONGVARCHAR, TINYINT, SMALLINT, INTEGER, BIGINT, FLOAT, REAL, DECIMAL, NUMERIC, NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Convert Functions

The convert functions supported in Composite are:

- CAST
- FORMAT_DATE
- PARSE_DATE
- PARSE_TIME
- PARSE_TIMESTAMP
- TO_NUMBER
- TO_TIMESTAMP
Of these, `CAST` is the only function that takes two arguments.

**CAST**

Given a valid expression and a target data type, this function converts the expression into the specified data type.

**Syntax**

```
CAST(expression AS target_data_type)
```

where, `expression` is the expression to be converted to the type indicated by `target_data_type`.

**Remarks**

- For acceptable input expressions and corresponding target data types, see Table 18, "Input and output types for CAST" on page 41.
- The output type is the same as that of the target data type except when the input expression is `NULL` or an empty string. If the input expression is `NULL` or an empty string, the output is of the same type as the input.
- Target data types **may** include length, precision, and scale arguments.

Example: `CAST(Orders_Qry.ShipPostalCode AS CHAR(5))`

- You can use `BLOB` or `CLOB` data types in this function.

Examples:

```
CAST(myBlob AS VARBINARY(size))
CAST (myVarBinary AS BLOB)
CAST (myClob AS VARCHAR(size))
CAST (myVarChar AS CLOB)
```

- When converting a `DECIMAL` to an `INTEGER`, the resulting value is truncated (For example, 15.99 will be converted to 15.)

- The `CAST` function can truncate strings without issuing an error. But it will give a runtime error if you cast the string '30000' to a `TINYINT`. If you cast an integer column to `TINYINT` and all the values are legal `TINYINT` values, you will not get an error. The `TO_CHAR` function is a special case and equates to `"CAST(x AS CHAR(255))"`. 
• All INTERVALs may be cast to CHAR and VARCHAR and vice versa.
  Interval days/hour/minute/seconds can only be cast to another interval
days/hour/minute/seconds. Interval year/month can only be cast to another interval
year/month. They are not interchangeable. Errors will be thrown if any data loss
occurs, as shown here:
  
  ```
  CAST(INTEGER '23' MONTH AS INTERVAL YEAR) = <error - 11
  months lost>
  CAST(INTEGRVAL '23' MONTH AS VARCHAR) = '23'
  CAST(INTEGRVAL '10' YEAR AS INTERVAL MONTH(3)) = INTERVAL '120'
  MONTH(3)
  ```

**Example**
```
SELECT
  products.UnitPrice, CAST(products.UnitPrice AS INTEGER) Price
FROM
  /shared/examples/ds_inventory/products products
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>TINYINT</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Input Argument 1 Type</td>
<td>Input Argument 2 Type</td>
<td>Output Type</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>SMALLINT</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>INTEGER</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>
### Table 18. Input and output types for CAST

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>BIGINT</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>FLOAT</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>FLOAT</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
### Table 18. Input and output types for CAST

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR CHAR VARCHAR LONGVARCHAR TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL NULL</td>
<td>REAL</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR CHAR VARCHAR LONGVARCHAR TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL NULL</td>
<td>DECIMAL</td>
<td>Same as that of argument 2.</td>
</tr>
</tbody>
</table>
### Table 18. Input and output types for CAST

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR VARCHAR VARCHAR TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL</td>
<td>NUMERIC</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR VARCHAR VARCHAR TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL DATE TIME TIMESTAMP</td>
<td>CHAR</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td>CAST</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Table 18. Input and output types for CAST

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>VARCHAR</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td>TINYINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td></td>
<td>LONGVARCHAR</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td>VARCHAR</td>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
### Table 18. Input and output types for CAST

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>STRING</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAST</td>
<td>DATE</td>
<td>DATE</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>TABLE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>TIME</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>TIME</td>
<td>NULL</td>
</tr>
<tr>
<td>CAST</td>
<td>CHAR</td>
<td>TIMESTAMP</td>
<td>Same as that of argument 2.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LONGVARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>TIMESTAMP</td>
<td>NULL</td>
</tr>
</tbody>
</table>
FORMAT_DATE

The FORMAT_DATE function accepts two arguments. The first argument must be a DATE, TIME, or TIMESTAMP. The second argument must be a string. The output is a VARCHAR(255). The output is a string representation of the DATE, TIME, or TIMESTAMP argument based on the format of the second argument.

Syntax

\texttt{FORMAT\_DATE(DATE|TIME|TIMESTAMP,FORMAT\_STRING)}

Remarks

- If the output exceeds 255 characters, it will be truncated.
- If the first argument is a DATE, the format string must not contain any TIME elements such as hour, minute, and seconds.
- If the first argument is a TIME, the format string must not contain any DATE elements such as year, month, and day of month.
- The format string is generally not case sensitive. Exceptions are noted in the following table, which lists the format string types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST</td>
<td>CHAR VARCHAR LONGVARCHAR TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL NULL</td>
<td>NULL</td>
<td>Same as that of argument 2.</td>
</tr>
</tbody>
</table>
### Table 19. Format String Types Supported in Composite

<table>
<thead>
<tr>
<th>Format String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fm</td>
<td>Fill mode. If this is used at the start of format, excess zeroes are suppressed.</td>
</tr>
<tr>
<td>yyyy</td>
<td>4-digit year (‘2006’)</td>
</tr>
<tr>
<td>yy</td>
<td>2-digit year (‘06’)</td>
</tr>
<tr>
<td>MONTH</td>
<td>Full month name (‘JULY’). Case is matched.</td>
</tr>
<tr>
<td>Mon</td>
<td>Abbreviated month name (‘JUL’). Case is matched.</td>
</tr>
<tr>
<td>mm</td>
<td>Numeric month (‘07’ or ‘7’ if fill mode).</td>
</tr>
<tr>
<td>DAY</td>
<td>Name of day (‘FRIDAY’). Case is matched.</td>
</tr>
<tr>
<td>Dy</td>
<td>Abbreviated name of day (‘FRI’). Case is matched.</td>
</tr>
<tr>
<td>dd</td>
<td>Day of month (‘04’ or ‘4’ if fill mode).</td>
</tr>
<tr>
<td>hh</td>
<td>12 Hour (‘11’)</td>
</tr>
<tr>
<td>hh24</td>
<td>24 Hour (‘23’)</td>
</tr>
<tr>
<td>AM</td>
<td>24 Hour (‘23’) Either results in the proper AM/PM string. Case is matched.</td>
</tr>
<tr>
<td>PM</td>
<td>24 Hour (‘23’) Either results in the proper AM/PM string. Case is matched.</td>
</tr>
<tr>
<td>mi</td>
<td>Minute (‘59’)</td>
</tr>
<tr>
<td>ss</td>
<td>Second (‘59’)</td>
</tr>
<tr>
<td>ff</td>
<td>Fractional seconds to millisecond level. (‘023’ or ‘23’ if fill mode)</td>
</tr>
</tbody>
</table>
Any leading whitespace will cause a misparse. Spaces, tabs, and newlines and the punctuation marks `/,.` are acceptable and will be passed onto the output. Characters may be enclosed in single quotes (like, 'quoted') to be passed directly to the output. The quotes will be removed. Use two single quotes in a row to output a single quote.

**Examples**

```sql
FORMAT_DATE(DATE '2000-02-01', 'Mon mon MON Month month MONTH')
```

will result in: Feb feb FEB February February FEBRUARY

```sql
FORMAT_DATE(DATE '2001-02-03', 'fmdd')
```

will result in: 3

```sql
FORMAT_DATE(TIME '23:59:01', 'hh hh24:mi:ss')
```

will result in: 23:59:01

**PARSE_DATE**

The function `PARSE_DATE` outputs a DATE by parsing the first argument using the second argument’s format.

The first argument must be a CHAR or VARCHAR. The second argument must also be a CHAR or VARCHAR, and must follow the same string format as the `FORMAT_DATE` function. The format string must not contain any non-date elements such as hours, minutes, or seconds.

**Syntax**

```sql
PARSE_DATE(data_string,format_string)
```

**Remark**

- When the two-digit year format ‘yy’ is used as the format string, 50 will result in 1950, and 49 will result in 2049.
**Examples**

<table>
<thead>
<tr>
<th>Function</th>
<th>Format String</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSE_DATE</td>
<td>'MARCH 06, 49', 'MONTH dd, yy'</td>
<td>2049-03-06</td>
</tr>
<tr>
<td>PARSE_DATE</td>
<td>'JAN 06, 2007', 'MONTH dd, yyyy'</td>
<td>2007-01-06</td>
</tr>
<tr>
<td>PARSE_DATE</td>
<td>'MARCH 06, 50', 'MONTH dd, yy'</td>
<td>1950-03-06</td>
</tr>
</tbody>
</table>

**PARSE_TIME**

The function `PARSE_TIME` is similar to `PARSE_DATE` except that the output of `PARSE_TIME` is a `TIME`.

**Syntax**

```
PARSE_TIME(data_string, format_string)
```

**Remarks**

- The format string must not contain any `DATE` elements such as year, month, and day of month.

**Example**

```
PARSE_TIME('23:59:31', 'hh24:mi:ss')
```

will result in a `TIME` value of: 23:59:31

**PARSE_TIMESTAMP**

The function `PARSE_TIMESTAMP` is similar to `PARSE_DATE` except that the output of `PARSE_TIMESTAMP` is a `TIMESTAMP`.

**Syntax**

```
PARSE_TIMESTAMP(data_string, format_string)
```
Examples

PARSE_TIMESTAMP('MARCH 06, 1923 03:59:31 am', 'MONTH dd, yyyy hh:mm:ss am')
will result in a TIMESTAMP value of: 1923-03-06 03:59:31

PARSE_TIMESTAMP('MARCH 06, 1923 23:59:31', 'MONTH dd, yyyy hh24:mi:ss')
will result in a TIMESTAMP value of timestamp: 1923-03-06 23:59:31

PARSE_TIMESTAMP('MARCH 06, 1923 23:59:31', 'MONTH dd, yyyy hh24:mi:ss')
will result in a TIME value of TIMESTAMP: 1923-03-06 23:59:31

TO_NUMBER

Converts a given string expression into a number.

Syntax

TO_NUMBER(expression)
where expression is a column name that returns a string, string literal, or the result of another function.

Remarks

- The input type is listed in Table 20, "Input and output types for TO_NUMBER" on page 53.
- The output is a decimal for non-null input values.
- If the input is NULL or an empty string, the output is the same as the input.
TO_TIMESTAMP

Converts a valid TIMESTAMP format into a valid TIMESTAMP format.

**Syntax**

```
TO_TIMESTAMP(expression)
```

where `expression` is a string.

**Remarks**

- The input cannot be an empty string. The input string is of a valid date/time format (YYYY MM DD HH:MM:SS) as follows:
  - `TO_TIMESTAMP('2003 10 12 15:59:59')`
  - `TO_TIMESTAMP('2003/10/12 15:59:59')`
  - `TO_TIMESTAMP('2003-10-12 15:59:59')`

- The input string should not have any leading zeros (0 or 00) in the year component.

---

Table 20. Input and output types for TO_NUMBER

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_NUMBER</td>
<td>CHAR, VARCHAR, LONGVARCHAR, STRING</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>
The following input strings are legal:

- `TO_TIMESTAMP('0')` returns `NULL`
- `TO_TIMESTAMP('00')` returns `2000-01-01 00:00:00`
- `TO_TIMESTAMP('0000')` returns `1999-12-01 00:00:00`
- `TO_TIMESTAMP('0000 00 00 00:00:00')` returns `2-11-30 00:00:00`
- `TO_TIMESTAMP('0000/00/00 00:00:00')` returns `2-11-30 00:00:00`
- `TO_TIMESTAMP('0000-00-00 00-00-00')` returns `2-11-30 00:00:00`

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TO_TIMESTAMP</code></td>
<td>CHAR, DATE, VARCHAR, LONGVARCHAR, STRING, TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

### Date Functions

The `Date` functions supported in Composite are:

- `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `DAY`, `MONTH`, and `YEAR`, `EXTRACT`, and `UTC_TO_TIMESTAMP`.

Of these, `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP` take no arguments, and the others take one argument.

**CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP**

These functions accept no arguments, and return the current date, current time, and the current date and time from the system clock of the machine where the database is running.
**Syntax**

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP

**Remarks**

- The output of `CURRENT_DATE` has the format: `YYYY-MM-DD`
- `CURRENT_TIME` takes an optional parameter, which is an unsigned integer that specifies the number of digits following the decimal point in the `SECONDS` field of `CURRENT_TIME`'s output. The output of `CURRENT_TIME` has the format: `HH:MM:SS.MS`
- `CURRENT_TIMESTAMP` takes an optional integer parameter specifying the number of significant digits. Composite SQL's time values default to Java precision ("3" - three significant digits, milliseconds).

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_DATE()</td>
<td>none</td>
<td>DATE</td>
</tr>
<tr>
<td>CURRENT_TIME()</td>
<td>none</td>
<td>TIME</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP()</td>
<td>INTEGER</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>

**DAY, MONTH, and YEAR**

These functions take a date expression as input, and returns the day, month, and year respectively from the date expression.

**Syntax**

DAY(date expression)
MONTH(date expression)
YEAR(date expression)
Remarks

- The input string cannot be an empty string.
- Leading zero in a date and month is ignored in the output.
- If the input is NULL, the output is also NULL.

Example

```sql
SELECT
  DAY(orders.OrderDate) OrderDate,
  MONTH(orders.OrderDate) OrderMonth,
  YEAR(orders.OrderDate) OrderYear
FROM
  /shared/examples/qs_orders/orders orders
```

Table 23. Input and output types for DAY, MONTH, YEAR

<table>
<thead>
<tr>
<th>Name and Format</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAY (date expression)</td>
<td>DATE TIMESTAMP</td>
<td>INTEGER</td>
<td>Output value is between 1 and 31.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>MONTH (date expression)</td>
<td>DATE TIMESTAMP</td>
<td>INTEGER</td>
<td>Output value is between 1 and 12.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>YEAR (date expression)</td>
<td>DATE TIMESTAMP</td>
<td>INTEGER</td>
<td>Output value is between 1 and 9999.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

EXTRACT

The EXTRACT function extracts a single field from a DATETIME or INTERVAL value.
Syntax

```
EXTRACT (<field name> FROM <value>)
```

where

- `<field name>` is `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, or `SECOND` and `<value>` is of type `DATETIME` or `INTERVAL`

**Example of EXTRACT with INTERVAL**

```sql
SELECT
    orders.OrderDate,
    EXTRACT(SECOND FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
    EXTRACT(MINUTE FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
    EXTRACT(HOUR FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
    EXTRACT(DAY FROM INTERVAL '2 23:51:19.124' DAY TO SECOND),
    EXTRACT(YEAR FROM INTERVAL '499-11' YEAR(3) TO MONTH),
    EXTRACT(MONTH FROM INTERVAL '500' MONTH(3))
FROM
    /shared/examples/ds_orders/orders
```

Results of the EXTRACT functions:

```
EXTRACT(SECOND FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 19.124
EXTRACT(MINUTE FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 51
EXTRACT(HOUR FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 23
EXTRACT(DAY FROM INTERVAL '2 23:51:19.124' DAY TO SECOND) = 2
EXTRACT(YEAR FROM INTERVAL '499-11' YEAR(3) TO MONTH) = 19.124
EXTRACT(MONTH FROM INTERVAL '500' MONTH(3)) = 500
```
Example of EXTRACT without INTERVAL

```sql
SELECT
    orders.ShipName,
    orders.OrderID,
    orders.OrderDate,
    EXTRACT(DAY FROM orders.OrderDate) "day",
    EXTRACT(MONTH FROM orders.OrderDate) "month"
FROM
    /shared/examples/ds_orders/orders orders
```

Remarks

- The data type of the output is an exact `NUMERIC` with a precision equal to the leading precision of `value` and a scale of 0 (zero). When the field name is a `SECOND`, the precision is equal to the sum of the leading precision and the seconds precision of `value` and a scale equal to the `SECOND`'s precision.
- When `value` is a negative `INTERVAL`, the result is a negative value.
- If `value` is `NULL`, the result is also `NULL`.

UTC_TO_TIMESTAMP

UTC refers to *Coordinated Universal Time*. The function `UTC_TO_TIMESTAMP` accepts one decimal or integer expression as the argument, and returns the timestamp. The unit measure of the argument is `seconds`.

The timestamp will be the number of seconds from UTC 00:00:00 January 1, 1970. If Composite Server is not in timezone GMT+0, the result from this function will be offset by the difference in the number of hours from GMT+0.

**Syntax**

```sql
UTC_TO_TIMESTAMP(DECIMAL|INTEGER)
```

Remarks

- If the input is `NULL`, the result will be `NULL`.
- The argument should not exceed 9223372036854775, and must not be less than -9223372036854775. Otherwise an exception will occur.
Example

\texttt{UTC\_TO\_TIMESTAMP(0)}

will return \texttt{TIMESTAMP 1970-01-01 00:00:00} if Composite Server is in timezone GMT+0

Numeric Functions

The Numeric functions supported in Composite are: \texttt{ABS, ACOS, ASIN, ATAN, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, PI, POWER, RADIANS, ROUND, SIN, SQRT, and TAN.} Of these, all the functions except \texttt{POWER} take one argument, whereas \texttt{POWER} takes two.

The following set of tables lists the input and output types for the \texttt{NUMERIC} functions.

\textbf{ABS}

Returns the absolute value of a given numeral. If the input is \texttt{NULL}, the output is also \texttt{NULL}.

Table 24. Input and output types for \texttt{ABS}

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ABS}</td>
<td>TINYINT, SMALLINT, INTEGER, BIGNUMERIC, FLOAT, REAL, DECIMAL</td>
<td>Same as that of the input.</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>\texttt{ABS}</td>
<td>INTERVAL</td>
<td>\texttt{INTERVAL}</td>
</tr>
<tr>
<td></td>
<td>\texttt{ABS(- INTERVAL '1' DAY) = INTERVAL '1' DAY}</td>
<td></td>
</tr>
</tbody>
</table>


ACOS

Returns the ACOS value of a given float expression.
The input value can range between -1 and +1, and the output is a float value. If the input is NULL, the output is also NULL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Input value range is: between -1.0 and +1.0</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASIN

Returns the ASIN value of a given float expression.
The input value can range between -1 and +1, and the output is a float value. If the input is NULL, the output is also NULL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIN</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Input value range is: between -1.0 and +1.0</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ATAN

Returns the ATAN value of a given numeral.

The output can range between -\( \pi/2 \) and \( \pi/2 \), and the output is a float value. If the input is NULL, the output is also NULL.

Table 27. Input and output types for ATAN

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
</table>
| ATAN | TINYINT  
      | SMALLINT  
      | INTEGER  
      | BIGINT  
      | NUMERIC  
      | FLOAT  
      | REAL  
      | DECIMAL  |
| NULL | NULL |

CEILING

Given a numeral, this function returns the smallest integer which is greater than or equal to the given numeral. If the input is NULL, the output is also NULL.

Table 28. Input and output types for CEILING

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
</table>
| CEILING | TINYINT  
         | SMALLINT  
         | INTEGER  
         | BIGINT  
         | FLOAT  
         | REAL  
         | DECIMAL  
         | NUMERIC |
| NULL | NULL |
COS

Returns the cosine value of a given numeral as a float. If the input is NULL, the output is also NULL.

Table 29. Input and output types for COS

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>COS</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Input value range is:</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td>between -1.0 and +1.0</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

COT

Returns the COTANGENT value of a given numeral as a float. If the input is NULL, the output is also NULL.

Table 30. Input and output types for COT

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COT</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
DEGREES

Given an angle in radians, this function returns the corresponding angle in degrees as a float. If the input is NULL, the output is also NULL.

Table 31. Input and output types for DEGREES

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEGREES</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>

EXP

Returns the exponential value of a given float expression as a float. If the input is NULL, the output is also NULL.

Table 32. Input and output types for EXP

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>
**FLOOR**

Given a numeral, this function returns the largest integer which is less than or equal to the given numeral. If the input is NULL, the output is also NULL.

**Table 33. Input and output types for FLOOR**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOOR</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**LOG**

Returns the logarithm of a given numeral as a float. If the input is NULL, the output is also NULL.

**Table 34. Input and output types for LOG**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Input value should be greater than 0 (zero).</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PI

Returns the value of π as a float (3.141592653589793). The function syntax is `PI()`. 

POWER

Given two numerals, this function returns the value of the first number raised to the power indicated by the second number, as a float. If any of the input arguments is NULL, the output is also NULL.

Table 35. Input and output types for POWER

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>TINYINT</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
RADIANS

Given an angle in degrees, this function returns the corresponding angle in radians as a float. If the input is NULL, the output is also NULL.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RADIANS</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

ROUND

Given two numerals, this function returns the value of the first number rounded to the value specified by the second number (that is, the scale). If the input is NULL, the output is also NULL.
Table 37. Input and output types for ROUND

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND</td>
<td>TINYINT</td>
<td>DECIMAL (a)</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL (p,q)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If scale is >= 0, the function works as follows:

- \( \text{ROUND}(\text{DECIMAL} (p,q), \text{scale}) \rightarrow \text{DECIMAL} \)
- \( p=q+\text{scale} \)
- \( \text{ROUND}(\text{INTEGER}, \text{scale}) \rightarrow \text{DECIMAL} \)
- \( 19+\text{scale} \)
- \( \text{ROUND}(\text{FLOAT}, \text{scale}) \rightarrow \text{DECIMAL} \)
- \( 255, \text{scale} \)
- \( \text{ROUND}(\text{STRING}, \text{scale}) \rightarrow \text{DECIMAL} \)
- \( 255, \text{scale} \)

If scale is < 0, treat scale as zero, and apply the above rules.

If scale is > 255, it is reduced to 255.

If scale is not a literal, thus unknown, treat scale as 4, which means that any value > 4 will be rounded down to 4 digits at runtime. Values < 4 will be rounded to the lesser number of digits, and zeros will be appended to fill it back to 4.

| NULL  | NULL               |             |
SIN

Returns the sine value of a given numeral as a float. If the input is NULL, the output is also NULL.

Table 38. Input and output types for SIN

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Output value range is: between -1.0 and +1.0</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQRRT

Returns the square root of a given numeral as a float. If the input is NULL, the output is also NULL.

Table 39. Input and output types for SQRRT

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQRRT</td>
<td>TINYINT</td>
<td>FLOAT</td>
<td>Output value range is: greater than or equal to 0 (zero).</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TAN

Returns the TANGENT of a given numeral as a float. If the input is NULL, the output is also NULL.

Table 40. Input and output types for TAN

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAN</td>
<td>TINYINT SMALLINT INTEGER BIGINT NUMERIC FLOAT REAL DECIMAL</td>
<td>FLOAT</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

XML Functions

This section describes the following 2GL functions supported in Composite:

- XMLAGG
- XMLATTRIBUTES
- XMLCONCAT
- XMLDOCUMENT
- XMLELEMENT
- XMLFOREST
- XMLNAMESPACES
- XMLQUERY
- XMLTEXT
- XPATH
- XSLT

The following functions are a part of the ANSI specification, but are not supported in Composite 4.0:

- XMLCOMMENT
- XMLTABLE
- XMLITERATE
- XMLBINARY
- XMLCAST
- XMLEXISTS
- XMLPARSE
- XMLPI
- XMLSERIALIZE
- XMLVALIDATE

XMLAGG

See “XMLAGG” on page 15.

XMLATTRIBUTES

The 2GL ATTRIBUTES function constructs XML attributes from the arguments provided. The result is an XML sequence with an attribute node for each input value.
Syntax

```xml
XMLATTRIBUTES <left paren> <XML attribute list> <right paren>
```

where XML attribute list is:

```xml
<XML attribute> [ { <comma> <XML attribute> }... ]
```

and

```xml
<XML attribute> is:
<XML attribute value> [ AS <XML attribute name> ]
```

and

```xml
<XML attribute value> is:
<value expression>
```

and

```xml
<XML attribute name> is
<identifier>
```

Example

```sql
SELECT
  XMLELEMENT(name Details, XMLATTRIBUTES (ProductID as product),
             XMLELEMENT(name orderno, OrderID),
             XMLELEMENT(name status, Status),
             XMLELEMENT(name price, UnitPrice)) myOutput
FROM /shared/examples/ds_orders/orderdetails
WHERE ProductID < 20
```

Remarks

- This function can only be used as an argument of the XMLELEMENT function.
- This function requires the AS keyword if aliases are used. This is in contrast to the select-list, which does not require the AS keyword for aliasing.
- This function cannot be used to insert blank spaces or new line characters.
Any `<value expression>` that evaluates to `NULL` will be ignored.
- All `<value expression>` must have an unique attribute names.
- If the result of every `<value expression>` is `NULL`, the result is `NULL`.

**XMLCONCAT**
This function concatenates one or more XML fragments together.

**Syntax**
```plaintext
XMLCONCAT <left paren> <XML value expression>
{ <comma> <XML value expression> }...
[ <XML returning clause> ] <right paren>
```

**Example**
```sql
SELECT
  XMLCONCAT(XMLTEXT(customers.ContactFirstName), XMLTEXT(' '),
             XMLTEXT(customers.ContactLastName)) AS CustomerName
FROM /shared/examples/ds_orders/customers customers
```

**Remarks**
- If an argument evaluates to `NULL`, that argument is ignored.
- If all arguments are `NULL`, the result is `NULL`.
- If only one non-`NULL` argument is supplied, the result of the function is that argument.

**XMLDOCUMENT**
This function generates an XML value with a single XQuery document node. It is equivalent to running the XQUERY expression.
Syntax

XMLDOCUMENT <left paren> <XML value expression>
[ <XML returning clause> ] <right paren>

where, <XML value expression> is:

a sequence of nodes of atomic values.

Example

SELECT
    XMLDOCUMENT
        XMLELEMENT(name Details, XMLATTRIBUTES{ProductID as product}),
        XMLELEMENT(name orderno, OrderID),
        XMLELEMENT(name status, Status),
        XMLELEMENT(name price, UnitPrice)) myXMLDocument
FROM /
    /shared/examples/ds_orders/orderdetails
WHERE
    ProductID < 20

XMLELEMENT

This function creates an XML node with an optional XML attributes node.

Syntax

XMLELEMENT <left paren> NAME <XML element name>
[ <comma> <XML namespace declaration> ] [ <comma> <XML attributes> ]
[ { <comma> <XML element content> }... ]
[ OPTION <XML content option> ]
[ <XML returning clause> ] <right paren>
Functions

where, the first argument is the name of the XML node. The name may be escaped if it contains certain characters. See section “Identifier Escaping” on page 77 for more details. The second optional argument is the `XMLNAMESPACE` function. The third optional argument is the `XMLATTRIBUTES` function. The fourth optional argument will be the content of the XML node. It may be an XML, numeric, or character type.

**Example**

```sql
SELECT
    XMLELEMENT(name Details, XMLATTRIBUTES(ProductID as product),
                XMLELEMENT(name orderno, OrderID),
                XMLELEMENT(name status, Status),
                XMLELEMENT(name price, UnitPrice)) myOutput
FROM
    /shared/examples/ds_orders/orderdetails
WHERE
    ProductID < 20
```

**Remarks**

- The element name may be escaped if it contains certain characters. For further details, see the section “Identifier Escaping” on page 77.
- If the XML element content evaluates to a character literal, it will be escaped. For details, see the section “Text Escaping” on page 78.

**XMLFOREST**

This function accepts one or more arguments and creates a series of XML nodes with the arguments being the children of each node.

**Syntax**

```sql
XMLFOREST <left paren> [ <XML namespace declaration> <comma> ]
    <forest element list>
    [ OPTION <XML content option> ]
    [ <XML returning clause> ]
    <right paren>
```
Example

```
SELECT
    XMLFOREST(CompanyName AS name, City AS city) as NameAndCityOfCompany
FROM
    /shared/examples/ds_orders/customers
```

Remarks

- Each argument to `XMLFOREST` can be followed by an optional alias. That alias will be the name of the XML node and the argument will be a child of that node.
- If no alias is specified and the argument is a column, the name of the column will be the name of the XML node.
- If the argument is not a column, an error will be generated.
- If the argument evaluates to a character literal, the resulting string will be escaped.

XMLNAMESPACES

Namespaces in XML provide a simple way to distinguish names used in XML documents. The `XMLNAMESPACES` function constructs namespace declarations from the provided arguments. This declaration can only be used as an argument for specific functions such as `XMLELEMENT` and `XMLFOREST`. The result is one or more XML namespace declarations containing in-scope namespaces for each non-null input value.

Example

```
SELECT
    CustomerID,
    XMLELEMENT(NAME customerName,
        XMLNAMESPACES('http://localhost:9400/services/webservices/ws/TestService/TestPort' AS "customers"),
        XMLATTRIBUTES(City AS city, ContactLastName as name))
"Customer Details"
```
FROM /services/webservices/ws/TestService/TestPort/customers
WHERE
   StateOrProvince = 'CA'

**XMLQUERY**

This function accepts one character literal argument, which is the XML query. Multiple arguments may be passed as input to the XML query. Each argument must be an XML data type or castable to an XML data type. Each argument can be followed by an optional identifier which gives the argument a variable name. If an argument is missing the identifier, the argument will become the context item. Only one context item per XMLQUERY function can exist. Each input must be resolved to an XML data type and must be aliased. Each alias must be unique, and is case sensitive.

**Syntax**

```
XMLQUERY <left paren>
   <XQuery expression>
   [ <XML query argument list> ]
   [ <XML returning clause> ]
   [ <XML query returning mechanism> ]
   [ <XML query empty handling option> ]
   <right paren>
```

**Remarks**

- Composite Server uses the Saxon as its XQUERY parser. Saxon requires that all XQUERY variables are declared as external variables in the XQUERY. This is not an ANSI requirement.
- Composite Server also requires all non context item variables to be declared in the xquery text. This is not ANSI specific. Variables can be declared through the format declare variable $<name> external;
   
   `<name>` is the name of the variable. Multiple declarations can be separated by a semicolon.
Currently, the XML passing mechanism is accepted but ignored. If the empty handling option is `NULL ON EMPTY`, `NULL` will be returned if the result of the XQuery is an empty element.

**Example**

```xml
XMLQuery('DECLARE variable $c EXTERNAL ; for $i in $c
  where $i /PDName = "Daniel Morgan"
  order by $i/PDName
  return $i/PDName' passing XMLELEMENT(name PDRecord, XMLELEMENT(name PDName, 'Daniel Morgan')) as c )
```

will result in: `<PDName>Daniel\ Morgan</PDName>`

**XMLTEXT**

This function accepts a character argument and returns the string after it has been escaped. See section “Text Escaping” on page 78 for more details.

**Syntax**

```xml
XMLTEXT <left paren> <character value expression> 
  [ <XML returning clause> ] <right paren>
```

**Example**

```sql
SELECT
  XMLTEXT(customers.CompanyName) as CompanyName,
  XMLTEXT(customers.City) as City
FROM
  /shared/examples/ds_orders/customers customers
```

**Remark**

- If the character argument evaluates to `NULL`, `NULL` is returned.
- The character value expression can accept `NULL`, `INTEGER`, `FLOAT`, `DECIMAL`, `DATE`, `TIMESTAMP`, `TIME`, `CLOB`, `BLOB`, `VARCHAR`, and `CHAR`. 

---

76
The XPATH function takes two arguments. The first argument is an XML value. The second argument is a string value containing an XPATH expression. The function evaluates the XPATH expression against the supplied XML value and returns the results as an XML value.

Example

```sql
PROCEDURE XpathFunctionExample(OUT resultXml XML)
BEGIN
    DECLARE sourceXml XML;
    DECLARE xpathExpression VARCHAR(4096);

    -- Create an XML value to use in the XPATH function.
    SET sourceXml = '<Book><Chapter>Test Data</Chapter></Book>\n
    -- Create an XPATH expression to evaluate.
    SET xpathExpression = '//Chapter';

    -- Evaluate the XPATH expression against the source XML value.
    SET resultXml = XPATH(sourceXml, xpathExpression);
END
```

Identifier Escaping

When creating XML nodes with XML elements, the name of the node may be escaped according to ANSI specification 9078-14 paragraph 4.10.3. The ANSI specification provides two modes of escaping: full escaping and partial escaping. Composite Server uses partial escaping. Only alpha characters and underscore may be leading characters. All other characters will be converted. Partially escaped identifiers escape all non leading non alpha numerical characters except minus (-), underscore (_), and colon (;) with the format `_xXXXX_` where `XXXX` is the
hexadecimal equivalent of the ASCII character. For example, the ampersand character & will be converted to _x0026_.

**Examples**

- XMLFOREST('black' AS 'a:') results in 'a:->black<a:->

**Text Escaping**

In an XML text, the following characters will be replaced as listed in the following table.

<table>
<thead>
<tr>
<th>Character in an XML Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>'</td>
<td>'</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

**Examples**

- XMLTEXT('a') is translated to &amp;

- XMLFOREST('>' AS green) is translated to &lt;green&gt;&gt;</green>

- XMLFOREST(NAME red, '') is translated to &lt;red&gt;&quot;&lt;/red&gt;
XSLT

The XSLT function takes two arguments. The first argument is an XML value. The second argument is a string value containing an XSLT expression. The function will evaluate the XSLT expression against the supplied XML value and return the results as an XML value.

Syntax

XSLT(sourceXml, xsltExpression)

Example

PROCEDURE XsltFunctionExample(OUT resultXml XML)
BEGIN
  DECLARE sourceXml XML;
  DECLARE xsltExpression VARCHAR(4096);

  -- Create an XML value to use in the XSLT function.
  SET sourceXml = '<Book><Chapter>Test Data</Chapter></Book>';

  -- Create an XSLT expression to evaluate.
  SET xsltExpression = '<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output omit-xml-declaration="true"/>
<xsl:strip-space elements="*"/>
<xsl:template match="/">
  <itemA>
    <xsl:for-each select="/Book">
      <itemB>
        <xsl:value-of select="Chapter"/>
      </itemB>
    </xsl:for-each>
  </itemA>
</xsl:template>
</xsl:stylesheet>'; -- Evaluate the XSLT expression against the source XML value.
  SET resultXml = XSLT(sourceXml, xsltExpression);
END
Operators

Composite supports the following types of operators:

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Condition Operators

The arithmetic operators are built-in which you can select from a proper Grid cell in the Studio, whereas you must manually type the other types of operators in your SQL.

Arithmetic Operators

The following arithmetic operators are built-in:

- ADD (+), DIVIDE (/), MODULO (%), MULTIPLY (*), NEGATE (-), and SUBTRACT (-)

Of these, all the operators except NEGATE take two arguments, whereas NEGATE takes only one.

The following section lists the input and output types for each of the built-in arithmetic operators.

Add (+)

Given two numerals, this function add them and returns the value.

Note on DECIMAL and NUMBER Types

If the input is DECIMAL or NUMERIC with any number datatypes other than DECIMAL or NUMERIC, the output datatype should be DECIMAL or NUMERIC with the same precision and scale as those of the DECIMAL or NUMERIC input.

Rules for DECIMAL or NUMERIC inputs:

- DECIMAL + DECIMAL -> DECIMAL
- DECIMAL + NUMERIC -> DECIMAL
- NUMERIC + DECIMAL -> DECIMAL
- NUMERIC + NUMERIC -> NUMERIC
The precision is the larger precision of the inputs plus 1 (one), and the scale is the larger scale of the inputs.
Example: DECIMAL(6,1) + NUMERIC(5,2) -> DECIMAL(7,2)

**Note on INTERVAL Type**

INTERVAL can be added to DATE, INTERVAL, TIME, or TIMESTAMP.
Interval days/hour/minute/seconds can only be added to another interval days/hour/minute/seconds, and interval year/month can only be added to another interval year/month. They are not interchangeable.

Rules for adding to an INTERVAL data type:

INTERVAL + DATE -> DATE
Only days, years, and months can be added to a DATE. When adding months, Composite will not round the day of month down and throws an error if the day of the month is invalid for the specified month.
INTERVAL + INTERVAL -> INTERVAL
INTERVAL + TIME -> TIME
INTERVAL + TIMESTAMP -> TIMESTAMP

Examples
DATE '1999-12-31' + interval '1' day = DATE '2000-01-01'
interval '1' month + DATE '1999-12-31' = DATE '2000-01-31'
DATE '1989-03-15' + interval '1' year = DATE '1990-03-15'
DATE '2000-01-31' + interval '1' month = <Error - February only has 28 days>
INTERVAL '6000' SECOND(4) + INTERVAL '3000' DAY(4) = INTERVAL '9000 01:40:00' DAY(4) TO SECOND
INTERVAL '6000' SECOND(4) + TIME '7:00:00' = TIME '08:40:00'

The plus symbol ‘+’ may also be used. The following expressions are all equivalent
+ INTERVAL '1' YEAR
INTERVAL '+1' YEAR
INTERVAL +'1' YEAR
### Table 42. Input and output types for + (plus)

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>FLOAT REAL</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>DECIMAL NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TINYINT SMALLINT INTEGER BIGINT STRING</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
Table 42. Input and output types for + (plus)

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (Add)FLOAT REAL</td>
<td>TINYINT SMALLINT INTEGER BIGINT STRING</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td>FLOAT REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td>DECIMAL NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td>DATE DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td>TIMESTAMP TIMESTAMP</td>
<td></td>
</tr>
<tr>
<td>+ (Add)DECIMAL NUMERIC</td>
<td>TINYINT SMALLINT INTEGER BIGINT</td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>FLOAT REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>DECIMAL NUMERIC</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>DATE DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>TIMESTAMP TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
Table 42. Input and output types for + (plus)

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (Add)</td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>STRING</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>STRING</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>Name</td>
<td>Input Argument 1 Type</td>
<td>Input Argument 2 Type</td>
<td>Output Type</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>+ (Add)</td>
<td>STRING</td>
<td>STRING</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td></td>
<td>FLOA T</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>NULL</td>
<td>TINYINT</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>FLOA T</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table 42. Input and output types for + (plus)
### Divide (/)

The matrix is similar to Multiply (*).

#### Note on DECIMAL and NUMERIC Types

If the input is DECIMAL or NUMERIC with any other number datatypes other than DECIMAL or NUMERIC, the output datatype should be DECIMAL or NUMERIC with the same precision and scale of the DECIMAL or NUMERIC input.

Rules for DECIMAL and NUMERIC inputs:

- DECIMAL / DECIMAL -> DECIMAL
- DECIMAL / NUMERIC -> DECIMAL
- NUMERIC / DECIMAL -> DECIMAL
- NUMERIC / NUMERIC -> NUMERIC

Formula for the output's precision and scale:

\[ \text{DECIMAL}(p_1, s_1) / \text{DECIMAL}(p_2, s_2) \rightarrow \text{DECIMAL}(p_1 + s_2, s_1 + p_2) \]

#### Note on the INTERVAL Type

INTERVAL may be divided by numbers.

- INTERVAL / NUMERIC -> INTERVAL

Example

- INTERVAL '90' HOUR / 10 = INTERVAL '09:00:00' DAY TO SECOND
- INTERVAL '1' YEAR / 1 = INTERVAL '01-00' YEAR TO MONTH

### Table 42. Input and output types for + (plus)

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (Add)</td>
<td>INTERVAL</td>
<td>DATE, INTERVAL, TIME, TIMESTAMP</td>
<td>Same as that or argument 2</td>
</tr>
</tbody>
</table>
### Modulo (%)
Given two numerals, this function returns the modulus after dividing the first number by the second.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>% (Modulo)</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Multiply (*)
Given two numerals, this function multiplies the two.

#### Note on DECIMAL and NUMERIC types
- If the input is DECIMAL/NUMERIC with any other number datatypes other than DECIMAL/NUMERIC, the output datatype should be DECIMAL/NUMERIC with the same precision and scale as those for the DECIMAL/NUMERIC input.

Rules for DECIMAL/NUMERIC inputs:
- DECIMAL(p1,s1) * DECIMAL(p2,s2) -> DECIMAL(p1+p2, s1+s2)
- DECIMAL(p1,s1) * NUMERIC(p2,s2) -> NUMERIC(p1+p2, s1+s2)
- NUMERIC(p1,s1) * DECIMAL(p2,s2) -> DECIMAL(p1+p2, s1+s2)
- NUMERIC(p1,s1) * NUMERIC(p2,s2) -> NUMERIC(p1+p2, s1+s2)
• The data type returned for a DECIMAL * INTEGER incorporates the size of the INTEGER into the resulting DECIMAL.
  
  Example:
  
  DECIMAL (p,s) * TINYINT -> DECIMAL (p+3, s)
  DECIMAL (p,s) * SMALLINT -> DECIMAL (p+5, s)
  
  **Note on the INTERVAL type**
  
  INTERVAL may be multiplied by numbers.
  
  INTERVAL * NUMERIC -> INTERVAL
  
  Example
  
  INTERVAL '1' DAY * 10 = INTERVAL '10 00:00:00' DAY TO SECOND
  INTERVAL '10' DAY * .1 = INTERVAL '1 00:00:00' DAY TO SECOND
  
  **Table 44. Input and output types for MULTIPLY**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (Multiply)</td>
<td>TINYINT</td>
<td>Same as that of argument 1.</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>FLOAT</td>
<td></td>
<td>FLOAT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>DECIMAL</td>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td>BIGINT</td>
<td>NUMERIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Input Argument 1 Type</td>
<td>Input Argument 2 Type</td>
<td>Output Type</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>* (Multiply) FLOAT REAL</td>
<td>TINYINT SMALLINT INTEGER BIGINT DECIMAL</td>
<td>Same as that of argument 1.</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT REAL</td>
<td>DECIMAL NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>FLOA T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* (Multiply) DECIMAL NUMERIC</td>
<td>TINYINT SMALLINT INTEGER BIGINT STRING</td>
<td>Same as that of argument 1.</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>FLOA T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL NUMERIC</td>
<td>REAL</td>
<td></td>
</tr>
</tbody>
</table>
Operators Chapter 1 SQL Support in Composite

Negate (-)

Returns the negative value of a given numeral.

Table 44. Input and output types for MULTIPLY

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (Multiply)</td>
<td>STRING</td>
<td>STRING</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NULL</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>NULL</td>
</tr>
<tr>
<td>STRING</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>NUMERIC</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td>NULL</td>
</tr>
<tr>
<td>* (Multiply)</td>
<td>INTERVAL</td>
<td>NUMERIC</td>
<td>INTERVAL</td>
</tr>
</tbody>
</table>
**Note on the INTERVAL Type**

INTERVAL can be negated.

The negative symbol (-) can be used, as in the following examples:

- INTERVAL '1' DAY
- INTERVAL '-1' DAY
- INTERVAL '-1' DAY

**Table 45. Input and output types for NEGATE**

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>(Negate) TINYINT</td>
<td>Same as that of the input.</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTERVAL</td>
<td></td>
</tr>
</tbody>
</table>

**Subtract** (-)

Given two numerals, this function subtracts the second one from the first.

**Note on the INTERVAL Type**

INTERVAL can be subtracted from DATE, INTERVAL, TIME, or TIMESTAMP.

Interval days/hour/minute/seconds can only be subtracted from another interval days/hour/minute/seconds, and interval year/month can only be subtracted from another interval year/month. They are not interchangeable.

INTERVAL can be subtracted from DATE, which results in a DATE.

Only days, years, and months can be subtracted from a DATE. When subtracting months, Composite will not round the day of month down and may throw an error if the day of the month is invalid for the specified month.
Operators

Chapter 1 SQL Support in Composite

Rules for adding to an INTERVAL data type:

- INTERVAL - DATE -> DATE
  Only days, years, and months can be subtracted from a DATE. When subtracting months, Composite will not round the day of month down and may throw an error if the day of the month is invalid for the specified month.

- INTERVAL - INTERVAL -> INTERVAL
- INTERVAL - TIME -> TIME
- INTERVAL - TIMESTAMP -> TIMESTAMP

Examples

- TIME '7:00:00' - INTERVAL '0 3:00:00' DAY TO SECOND = TIME '4:00:00'
- INTERVAL '10000-11' YEAR(5) TO MONTH - INTERVAL '1' MONTH(1) = INTERVAL '10000-10' YEAR TO MONTH
- DATE '1999-12-31' - INTERVAL '365' day(3) = DATE '1998-01-01'

Table 46. Input and output types for MINUS

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (Subtract)</td>
<td>TINYINT</td>
<td>TINYINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TINYINT</td>
<td>STRING</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>SMALLINT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIGINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Operators Chapter 1 SQL Support in Composite

Table 46. Input and output types for MINUS

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (Subtract) TINYINT</td>
<td>FLOAT</td>
<td></td>
<td>FLOAT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINYINT</td>
<td>DECIMAL (p,s)</td>
<td></td>
<td>DECIMAL (p, s)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>NUMERIC (p, s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If one input is DECIMAL(p,s)/NUMERIC(p,s) and the other is any other number other than DECIMAL/NUMERIC, the output of the SUBTRACT function should be DECIMAL(p,s)/NUMERIC(p,s).

For example, INTEGER - DECIMAL(5,1) -> DECIMAL(5,1).

If the two inputs are DECIMAL, the output’s precision is the larger on of the two inputs, and the same for scale.

For example, DECIMAL(6,1) - DECIMAL(5,2) -> DECIMAL(6,2).

If the inputs are DECIMAL and NUMERIC, the output is DECIMAL, and the output’s precision and scale follow the above rule.

If the inputs are NUMERIC and NUMERIC, the output is NUMERIC with the same rules for precision and scale.
<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (Subtract)</td>
<td>FLOAT</td>
<td>TINYINT</td>
<td>FLOAT</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLOAT</td>
<td>DECIMAL (p, s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REAL</td>
<td>DECIMAL (p, s)</td>
<td></td>
</tr>
<tr>
<td>- (Subtract)</td>
<td>DECIMAL</td>
<td>TINYINT</td>
<td>DECIMAL</td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMERIC</td>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
<td>DECIMAL</td>
<td></td>
</tr>
</tbody>
</table>
### Table 46. Input and output types for MINUS

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (Subtract)</td>
<td>DATE</td>
<td>DATE</td>
<td>An INTERVAL day. It is the number of days between the two arguments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DATE '2006-03-20' - DATE '2005-12-02' = INTERVAL '108' DAY(3)</td>
</tr>
<tr>
<td></td>
<td>DATE</td>
<td>TIMESTAMP</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>- (Subtract)</td>
<td>TIME</td>
<td>TIME</td>
<td>An INTERVAL hour to second.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TIME '21:00:00' - TIME '19:00:00' = INTERVAL '02:00:00' DAY TO SECOND</td>
</tr>
</tbody>
</table>
 Operators Chapter 1 SQL Support in Composite

The result represents the difference between the dates in the two inputs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Argument 1 Type</th>
<th>Input Argument 2 Type</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- (Subtract)</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>An INTERVAL day to second:</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP '2006-03-20 21:00:00' - TIMESTAMP '2005-12-02 19:00:00' = INTERVAL '108 02:00:00' DAY(3) TO SECOND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>STRING</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TINYINT</td>
<td>SMALLINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>INTEGER</td>
<td>BIGINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>NUMERIC</td>
<td>FLOAT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>REAL</td>
<td>DECIMAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATE</td>
<td>TIMESTAMP</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
Comparison Operators

The following comparison operators are not available through the Studio UI, so you must type them manually in your query:
- `=` (equal to)
- `<` (not equal to)
- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
(equal to)

```sql
SELECT
    ProductName, UnitPrice
FROM
    /shared/examples/ds_inventory/products products
WHERE
    ProductID = 5
```

(not equal to)

```sql
SELECT
    ProductName, UnitPrice
FROM
    /shared/examples/ds_inventory/products products
WHERE
    ProductID <> 10
```

(less than)

```sql
SELECT
    ProductName, UnitPrice
FROM
    /shared/examples/ds_inventory/products products
WHERE
    ProductID < 10
```

(greater than)

```sql
SELECT
    ProductName, UnitPrice
FROM
    /shared/examples/ds_inventory/products products
WHERE
    ProductID > 10
```
Operators

<= (less than or equal to)

```sql
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID <= 5
```

>= (greater than or equal to)

```sql
SELECT ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
WHERE ProductID >= 5
```

Quantified Comparisons

When a comparison operator (<, <=, >, >=, <>) is used together with the words ALL, ANY, or SOME, the comparison is known as being "quantified." ANY and SOME are equivalent.

Such comparisons operate on subqueries that could return multiple rows but would return a single column.

**Syntax**

```sql
<expression> <comparison-operator> {ALL | ANY | SOME} <column-subquery>
```

where the `<comparison-operator>` can be: >, <, >=, <=, or ==

**Remarks**

- ALL or ANY is applicable only to subqueries. When one of them is used, the comparison converts a scalar subquery to a column subquery.
- If ALL is used, the comparison must be true for all values returned by the subquery.
If **ANY** or **SOME** is used, the comparison must be true for at least one value of the subquery.

A subquery using **ANY** must return a single column. **ANY** compares a single value to the column of data values produced by the subquery.

If any of the comparisons yields a value of **TRUE**, the **ANY** comparison returns a value of **TRUE**.

If the subquery returns **NULL**, the **ANY** comparison returns **FALSE**.

**ALL** is used to compare a single value to the data values produced by the subquery. The specified comparison operator is used to compare the given value to each data value in the result set. If all of the comparisons returns a value of **TRUE**, the **ALL** test also returns **TRUE**.

If the subquery returns an empty result set, the **ALL** test returns a value of **TRUE**.

If the comparison test is false for any values in the result set, the **ALL** search returns **FALSE**.

The **ALL** search returns **TRUE** if all the values are true. Otherwise, it returns **UNKNOWN**. For example, if there is a **NULL** value in the subquery result set but the search condition is **TRUE** for all non-**NULL** values, the **ALL** test returns **UNKNOWN**.

Negating the **ALL** comparison. The following expressions are **not** equivalent.

```sql
NOT a = ALL (subquery)
```

```sql
a <> ALL (subquery)
```

**Example using ANY**

```sql
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (SELECT ShipDate
FROM SalesOrderItems
WHERE ID=500);
```

The above query returns the order ID and customer ID for the orders that were placed after at least one product with the order ID 500 was shipped.

You can use **SOME** instead of **ANY**, as in the following example:
Example using SOME

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (SELECT ShipDate
                           FROM SalesOrderItems
                           WHERE ID=500);
```

Example using ALL

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (SELECT ShipDate
                        FROM SalesOrderItems
                        WHERE ID=500);
```

In the above example, the main query tests the order dates for each order against the shipping dates of every product with the ID 500. If an order date is greater than the shipping date for every shipment with the order ID 500, then the ID and customer ID from the SalesOrders table are a part of the result set.

For further reading, see:


Logical Operators

The following logical operators are **not** built-in, so you must type them in your query:

- **AND**
- **NOT**
- **OR**

**AND**

```sql
SELECT ProductID, ProductName, ProductDescription
FROM /shared/examples/ds_inventory/products products
WHERE ReorderLevel > 5
AND LeadTime = '1 Day'
```

**NOT**

```sql
SELECT orderdetails.*
FROM /shared/examples/ds_orders/orderdetails orderdetails
WHERE NOT (UnitPrice > 100.00)
```

**Using two NOT conditions**

```sql
SELECT orderdetails.*
FROM
    /shared/examples/ds_orders/orderdetails orderdetails
WHERE
    NOT (UnitPrice > 100.00) AND NOT (Quantity < 2)
```
OR

SELECT
  ProductID, ProductName, ProductDescription
FROM
  /shared/examples/ds_inventory/products products
WHERE
  ReorderLevel > 5
OR
  UnitPrice > 22.00

Condition Operators

Composite supports the following condition operators: CASE, COALESCE, IN, IS NOT NULL, IS NULL, LIKE.

CASE

The CASE operator is used to evaluate several conditions and return one value for the first matched condition. It is similar to CASE, IF . . . THEN . . . ELSE, and SWITCH statements used in many programming languages. However, in SQL, CASE is an expression, not a statement.

CASE has two formats, simple syntax (Simple Syntax) and searched syntax (Searched Syntax). Simple CASE compares an expression to a set of simple expressions, whereas the searched CASE compares an expression to a set of logical expressions.

Simple Syntax

CASE <comparison-value>
  WHEN <conditional-expression 1> THEN <scalar-expression 1>
  WHEN <conditional-expression 2> THEN <scalar-expression 2>
  WHEN <conditional-expression 3> THEN <scalar-expression 3>
  [ELSE <default-scalar-expression>]
END
Remarks

- Using `CASE`, you can express an alternate value to an underlying value. For example, if the underlying value is a code (such as 1, 2, 3), you can display it as a humanly readable string value (Small, Medium, Large), without affecting the actual, underlying value (1, 2, 3).
- If none of the test conditions is true, `CASE` will return the result of `ELSE`, which is optional.
- If no match is found, and `ELSE` is not specified, `ELSE NULL` is assumed by default.

**Searched Syntax**

```sql
CASE
  WHEN <conditional-expression 1> THEN <scalar-expression 1>
  WHEN <conditional-expression 2> THEN <scalar-expression 2>
  WHEN <conditional-expression 3> THEN <scalar-expression 3>
  [ELSE <default-scalar-expression>]
END
```

**Example of simple `CASE`**

```sql
SELECT
  ProductID, Status, UnitPrice
CASE Status
  WHEN 'open'
    THEN UnitPrice * 1.10
  WHEN 'closed'
    THEN UnitPrice * 1
  ELSE UnitPrice
END
AS
  "New Price"
FROM
  /shared/examples/ds_orders/orderdetails
```
Examples of searched CASE expressions

```
SELECT ProductID, UnitPrice
CASE
  WHEN UnitPrice <= 100
  THEN 'Between $1 and $100.00'
  WHEN UnitPrice <= 200
  THEN 'Between $100.01 and $200.00'
ELSE 'Over $200.00'
END
  AS "Price Range"
FROM /shared/examples.ds_orders/orderdetails
```

```
SELECT ProductID, UnitPrice
CASE
  WHEN UnitPrice >= 301
  THEN 'Above 300.00'
  WHEN UnitPrice >= 400
  THEN 'Between 301.01 and 400.00'
ELSE 'Over $200.00'
END
  AS "Price Range"
FROM /shared/examples.ds_orders/orderdetails
```
COALESCE

COALESCE() returns the first non-null expression among its arguments.

Syntax

\[
\text{COALESCE}(\text{expr1, expr2, expr3...})
\]

is equivalent to

\[
\text{CASE}
\begin{align*}
\text{WHEN } \text{expr1} \text{ NOT NULL THEN } & \text{expr1} \\
\text{WHEN } \text{expr2} \text{ NOT NULL THEN } & \text{expr2} \\
\text{ELSE } & \text{expr3}
\end{align*}
\text{END}
\]

Example

\[
\text{SELECT}\]
\[
\text{CAST} (\text{COALESCE (hourly\_wage \ast 40 \ast 52, salary, commission \ast num\_sales)} \text{ AS money})
\]
\[
\text{FROM}
\]
\[
\text{wages}
\]

EXISTS

The EXISTS keyword is used to test the existence of specific rows in the result of a subquery, while the NOT EXISTS keyword is used to test the non-existence of specific rows in the result of a subquery.

Syntax (for EXISTS)

\[
\text{<source-expression>}
\text{WHERE EXISTS <subquery>}
\]

Syntax (for NOT EXISTS)

\[
\text{<source-expression>}
\text{WHERE NOT EXISTS <subquery>}
\]
Remarks

- `EXISTS` just checks for the existence of rows under specified conditions in the subquery, and the actual values in those rows are irrelevant. Therefore, the `SELECT` clause in the subquery is `SELECT *` to retrieve all the columns.
- The subquery can return any number of rows and columns.
- The subquery would return at least one row if the `EXISTS` condition is met. If the subquery returns at least one row, the `NOT EXISTS` condition is false. Conversely, if the subquery does not return any row, the `EXISTS` condition is not met and the `NOT EXISTS` condition is true.
- Even if the rows returned by the subquery contain `NULL` values, they are not ignored. Such rows are considered as normal rows.

Example for `EXISTS`

```
SELECT *
FROM /shared/examples/ds_inventory/suppliers
WHERE EXISTS (SELECT *
              FROM /shared/examples/ds_inventory/purchaseorders
              WHERE purchaseorders.SupplierID = 5)
```

Example for `NOT EXISTS`

```
SELECT *
FROM /shared/examples/ds_inventory/suppliers
WHERE NOT EXISTS (SELECT *
                   FROM /shared/examples/ds_inventory/purchaseorders
                   WHERE purchaseorders.SupplierID = 100)
```

The `IN` operator is used to determine whether a given value matches any value in a list of target values. The list of values can be generated using a subquery.
The `IN` operator has two formats (Syntax 1 and “Syntax 2” on page 110).

**Syntax 1**

```
<source-expression>
[NOT] IN <scalar-expression-list>
```

`IN` is a comparison operator like `<` (less than) or `LIKE` and is legal anywhere a conditional expression is used. That is, you can place `IN` in a `WHERE` clause, `HAVING` clause, or `JOIN` `ON` clause, as well as in a `CASE` expression. The above syntax uses the `WHERE` clause.

All the expressions in the target list, indicated by `<scalar-expression-list>` in the syntax, must be compatible or implicitly castable to the source expression, indicated by `<source-expression>` in the syntax or vice versa.

If the items in the target list are not all of the same type, as in the following example:

```
ID IN (1000, 'X', 12.0)
```

the list will be translated to the following format:

```
(left = right1) OR (left=right2) OR (left=right3)
```

with `CAST` functions as necessary.

**Remarks**

- You can use `IN` with data types that are comparable or implicitly castable to each other.
- You can combine `IN` conditions with `AND` and `OR` conditions.
- The expression “A IN (B, C)” is equivalent to the expression “A = B or A = C”.
- You can use `NOT IN` to negate the `IN` condition. That is, `NOT IN` will specify values that are not in the target list.

**Examples**

The following examples illustrate the use of `IN` with a string, number, `DATE`, `AND`, `OR`, and `NOT`. 
Using IN with a string

```sql
SELECT
    customers.CompanyName, customers.StateOrProvince
FROM
    /shared/examples/ds_orders/customers customers
WHERE
    StateOrProvince IN ('CA', 'PA')
```

Using IN with a number

```sql
SELECT
    Inventory.Model, Inventory.Make, Inventory."Year"
FROM
    composite.admin.ds_access@Inventory Inventory
WHERE
    "Year" IN ('1991')
```

Using IN with date

```sql
SELECT
    purchaseorders.ShipDate, SupplierID
FROM
    /shared/examples/ds_inventory/purchaseorders PurchaseOrders
WHERE
    ShipDate IN ( CAST('2003-02-06' AS DATE), CAST('2003-02-07' AS DATE) )
```
Using IN with AND and OR

SELECT
    purchaseorders.ShipDate, SupplierID
FROM
    /shared/examples/ds_inventory/purchaseorders PurchaseOrders
WHERE
    ShipDate IN (TO_DATE('2003-02-06'))
    AND
    ShippingMethodID = 3
    OR
    DatePromised = '2003-02-02'
    OR
    ShipDate IN ('2001-05-08', DATE '2001-04-01', '2000-02-25')

Syntax 2

<source-expression>
[NOT] IN <subquery>

Remarks

- The subquery, indicated by <subquery> in the syntax, can return only one column of a compatible data type. However, it is allowed to return multiple rows.
- The subquery is run once for the parent query (prior to running the parent query), to populate the list of values for the IN clause.
- See also the Remarks for “Syntax 1” on page 9.

Example

SELECT
    Customers.ContactName
FROM

110
Using NOT IN

```sql
SELECT Customers.ContactName, CompanyName
FROM /shared/examples/ds_orders/Customers Customers
WHERE City NOT IN
    (SELECT City FROM /shared/examples/ds_orders/Customers Customers
     WHERE City = 'New York')
```

IS NOT NULL

```sql
SELECT Employees.FirstName, Employees.LastName, Employees.WorkPhone
FROM /services/databases/ds_service/Employees Employees
WHERE BillingRate IS NOT NULL
```

IS NULL

```sql
SELECT Employees.FirstName, Employees.LastName, Employees.WorkPhone
FROM /services/databases/ds_service/Employees Employees
```
WHERE
    BillingRate
    IS NULL

LIKE

The LIKE operator is used to match strings based on a pattern. The pattern string can contain wildcard characters which have special meaning:
% (percentage symbol). Matches any sequence of zero or more characters.
_ (underscore). Matches any single character.

Syntax for the LIKE reserved word:

```
column LIKE pattern [ESCAPE escape-character]
```

where the ESCAPE clause is optional.

```
SELECT
    ProductID, ProductName, ProductDescription
FROM
    /shared/examples/ds_inventory/products products
WHERE ProductName LIKE 'Acme'
```

It matches Acme Memory, Acme Processor, and Acme Storage 40GB.

```
SELECT
    company, credit_limit
FROM
    customers
WHERE
    company LIKE 'Smiths_n Corp.'
```

It matches Smithson Corp., and Smithsen Corp.
If the data value in the column is null, the like test returns a NULL result.
You can locate the strings that do not match a pattern by using NOT LIKE.

**The ESCAPE Character**

The ESCAPE character is used to match the wild card characters themselves, as shown here.

```sql
SELECT
    order_num, product
FROM
    orders
WHERE
    product LIKE 'A$%BC%' ESCAPE '$'
```

The first % sign will not be treated as wild card character because of the $ escape character.
SQL Keywords

Composite supports the following SQL keywords (listed in alphabetical order):

- BETWEEN
- CROSS JOIN
- DELETE
- DISTINCT
- EXCEPT
- FULL OUTER JOIN
- GROUP BY
- HAVING
- INNER JOIN
- INSERT
- INTERSECT
- LEFT OUTER JOIN
- ORDER BY
- RIGHT OUTER JOIN
- SELECT
- UNION
- UNION ALL
- UPDATE
- WHERE

Using SQL Keywords

To access the SQL panel in the Modeler, click the SQL tab in the right-frame.

When you use the SQL panel for your query operations, the visual modeling, if any, that you have done for the current query using the Model area become invalid. Therefore, you can save your current query under a different name to be used later as needed, and use the SQL tab to modify and execute the current query, or you can use the Generate Model button on the SQL panel’s toolbar to generate the design corresponding to the current SQL.
**BETWEEN**

BETWEEN is a range filter. The BETWEEN range contains a low value and a high value. The low value must be less than or equal to the high value.

**Syntax**

```
WHERE test_column [NOT] BETWEEN low_value AND high_value
```

**Example 1**

```
SELECT ProductID, ProductName
FROM /shared/examples/ds_orders/products
WHERE UnitPrice BETWEEN 50 and 100
```

**Example 2**

```
SELECT OrderID
FROM /shared/examples/ds_orders/orders
WHERE OrderDate BETWEEN DATE '2003-02-03' AND DATE '2003-02-06'
```

**Notes**

- Both low and high values are included in the search.
- BETWEEN can be used in both WHERE and HAVING clauses.
- BETWEEN works with character strings, numbers and date times. Only the values that are identical to the search values will be returned.
- BETWEEN can be rewritten using <= and >=, as follows:
  
  ```
  WHERE test_column >= low_value AND test_column <= high_value
  ```
CROSS JOIN

CROSS JOIN takes the Cartesian product, that is, all combinations of each table in the join. For example, a CROSS JOIN involving two tables in which one table has 4 rows and the second table has 5 rows would result in 20 rows.

DELETE

Composite supports the regular SQL DELETE statement. See also “INSERT/UPDATE/DELETE on Views” on page 136.

Syntax

DELETE FROM <table>
[WHERE <criteria>]

The WHERE clause is optional. The rules for the WHERE clause of an UPDATE statement is the same as the rules for the WHERE clause of a SELECT statement.

Example 1

The following example deletes all the rows in the orders table:

```
DELETE FROM
/shared/examples/ds_orders/orders
```

Example 2

The following example deletes the row where the ProductID = 44 in the orders table:

```
DELETE FROM
/shared/examples/ds_orders/orders
WHERE
ProductID = 44
```

Example 3

The following example uses a subquery:

```
DELETE FROM
/shared/examples/ds_orders/orders
WHERE
DISTINCT

DISTINCT eliminates duplicate rows from the result set.

Example of DISTINCT

```sql
SELECT DISTINCT StateOrProvince
FROM /shared/examples/ds_orders/customers customers
```

Notes

- If any of the columns has a NULL value, it is treated as any other value.
- If you have DISTINCT and GROUP BY in your SELECT clause, the GROUP BY is applied first before DISTINCT.
- All data types (incl: BLOB, CLOB, and XML) are supported by DISTINCT.
- DISTINCT in the SELECT clause and DISTINCT in an aggregate function do not return the same result.

EXCEPT

EXCEPT is like the UNION statement, except that EXCEPT produces rows that result from the first query, but not the second.

EXCEPT is known as MINUS in Oracle.

EXCEPT ALL: If a row appears \(a\) times in the first table and \(b\) times in the second table, it will appear \(a - b\) or \(0\) (zero), whichever is greater.

EXCEPT: Similar to EXCEPT ALL and eliminates the duplicates.
Syntax
<query-expression>
EXCEPT [ALL]
<query-expression>

Example (EXCEPT)
The following query lists the cities where suppliers live but a customer does not.
```
SELECT
  City
FROM
  /shared/examples/ds_inventory/suppliers
EXCEPT
SELECT
  City
FROM
  /shared/examples/ds_orders/customers
```

Example (EXCEPT ALL)
```
SELECT
  City
FROM
  /shared/examples/ds_inventory/suppliers
EXCEPT ALL
SELECT
  City
FROM
  /shared/examples/ds_orders/customers
```

Notes
- Unlike UNION and INTERSECT, EXCEPT is not commutative. That is, A EXCEPT B is not the same as B EXCEPT A.
- The rules are the same as that of UNION.
**FULL OUTER JOIN**

*Example of FULL OUTER JOIN*

```
SELECT *
FROM /shared/examples/ds_orders/orderdetails orderdetails
FULL OUTER JOIN /shared/examples/ds_orders/products products
ON orderdetails.ProductID = products.ProductID
```

**GROUP BY**

GROUP BY is used when multiple columns from one or more tables are selected and at least one aggregate function appears in the SELECT statement. In that case, you need to GROUP BY all the selected columns except the one(s) operated on by the aggregate function.

All data types (incl: BLOB, CLOB, and XML) are supported by GROUP BY.

*Example of GROUP BY*

```
SELECT
  ORDERDETAILS.STATUS, count(orderdetails.status) as Item_Count
FROM /shared/examples/ds_orders/orderdetails orderdetails
INNER JOIN /shared/examples/ds_inventory/products Products ON
  orderdetails.ProductID = products.ProductID
INNER JOIN /shared/examples/ds_orders/orders Orders ON
  orders.OrderID = orderdetails.OrderID
```

119
GROUP BY
  orderdetails.Status

HAVING

The HAVING clause is used in combination with GROUP BY. You can use HAVING in a
SELECT statement to filter the records that a GROUP BY returns.

Example of HAVING

SELECT
  OrderID, SUM(orderdetails.Quantity) sumQuantity
FROM
  /shared/examples/ds_orders/orderdetails orderdetails
GROUP BY
  OrderID
HAVING
  SUM(orderdetails.Quantity) > 10

INNER JOIN

Example of INNER JOIN

SELECT
  products.ProductName, products.ProductID
FROM
  /shared/examples/ds_inventory/products products
  /shared/examples/ds_inventory/products products_1
INNER JOIN
  /shared/examples/ds_inventory/products products_1
ON
  products.ProductID = products_1.ProductID
The **INSERT** statement adds new rows into a table. You can insert a single row or multiple rows all at one time.

You can use an **INSERT** statement only in a SQL script or from a JDBC/ODBC call. See also "**INSERT/UPDATE/DELETE on Views**" on page 136.

The **INSERT INTO** statement may also be used to insert a complete row of values without specifying the column names. Values must be specified for every column in the table in the order specified by the DDL. If the number of values is not exactly the same as the number of columns in the table or if a value is not allowed for a particular data type, an exception will be thrown.

The **INSERT** statement itself does not return a result, but the database system returns a message informing how many rows have been affected. Then you can verify the insertion by querying the data source.

Currently, Composite supports the insert functionality in the following types of data sources:

- DataDirect - Adabas, DB2, IMSDB, Mainframe, VSAM, VSAM CICS
- Netezza
- DB2 V8
- Oracle EBS
- DB2 V7
- Oracle 8i
- Informix
- Oracle 9i
- File - Delimited
- Oracle 10g
- Microsoft Access
- PeopleSoft
- Microsoft Excel
- Salesforce
- Microsoft SQL Server
- Siebel
- MySQL 3.x
- Sybase
- MySQL 4.x
- Teradata
- Teradata
- Teradata
- Teradata
You cannot insert into the following types of resources:
- Custom Java Procedure
- File - XML
- LDAP
- WSDL

**Syntax 1**

```
INSERT INTO <table_name> DEFAULT VALUES
```

**Syntax 2**

```
INSERT INTO <table_name> [(<columnA, columnX, ...>)]
VALUES (<valueList>), (<valueList>)*
```

**Syntax 3**

```
INSERT INTO <table_name> [(<columnA, columnX, ...>)]
<queryExpression>
```

Opening and closing parenthesis are used for grouping. `<queryExpression>` indicates a `SELECT` statement.

Note that listing of the columns is optional, as indicated by the square bracket enclosure. In all cases, the number and type of the values must be equal and consistent with the number of columns in the row or as specified.

See “Multi-row INSERT with `<queryExpression>`” on page 124 for an example.

**Remarks**

- The system will automatically discard any `ORDER BY` in the sub queries, because it is not useful to sort the subquery.
- In a multi-row `INSERT`, the query result must contain the same number of columns in the same order as the column list in the `INSERT` statement and the data types must be compatible, column-by-column.
- If a non-nullable column is set to null, the data source will throw a runtime exception.
- `INSERT` statements should include all non nullable-columns
- Derived columns cannot be present in an `INSERT` statement
Examples
This section contains examples of different types of INSERTs.

Single row INSERT
PROCEDURE sc2()
BEGIN
    INSERT INTO
        /shared/examples/ds_inventory/products (ProductID, ProductName, UnitPrice)
    VALUES (23, 'monitor', 500.00);
END

Multi-row INSERT 1
PROCEDURE sc2()
BEGIN
    INSERT INTO
        /shared/examples/ds_inventory/products (ProductID, ProductName, UnitPrice)
    VALUES
        (41, 'monitor', 1000/10 * 1),
        (42, 'monitor', 1000/10 * 1),
        (43, 'monitor', 1000/10 * 1);
END
Multi-row INSERT with <queryExpression>

```
PROCEDURE get_open_orders(OUT numOpen INTEGER)
BEGIN
    -- Clear the table
    DELETE FROM /users/composite/test/sources/mysql/updates;

    -- Get all open orders
    INSERT INTO /users/composite/test/sources/mysql/updates
    (c_bigint, c_varchar)
    SELECT OrderID, Status
    FROM /shared/tutorial/sources/ds_orders/orderdetails
    WHERE Status = 'Open';

    -- Return number of open orders
    SELECT count(*) INTO numOpen
    FROM /users/composite/test/sources/mysql/updates;
END
```

INSERT with DEFAULT

```
INSERT INTO Customers (FirstName, LastName, Country)
VALUES ('joe', 'Ely', DEFAULT)
```

An exception will be thrown if the target database does not support DEFAULT keyword.
A runtime exception will be thrown if the column does not have a default defined and is non nullable.

INSERT with DEFAULT VALUES

```
INSERT INTO Customers DEFAULT VALUES
```
If `DEFAULT VALUES` clause is specified, a single row is inserted into a table containing the appropriate defaults (possibly null) in every column. It is an error if any column has no default.

**INTERSECT**

INTERSECT is like the `UNION` statement, except that INTERSECT produces rows that appear in both queries involved. INTERSECT ALL: If a row appears x times in the first table and y times in the second table, it will appear z times in the result table where z is the lesser of x and y. INTERSECT: Similar to INTERSECT ALL, and eliminates duplicate rows.

According to SQL standard, INTERSECT has higher precedence than UNION and EXCEPT.

**Syntax**

```sql
<query-expression>
INTERSECT [ALL] <query-expression>
```

**Remarks**

- The rules are the same as those for `UNION` (on page 132).

**Example (INTERSECT)**

The following query lists the cities where suppliers and customers are found.

```sql
SELECT City
FROM /shared/examples/ds_inventory/suppliers
INTERSECT
SELECT City
FROM /shared/examples/ds_orders/customers
```
**Example (INTERSECT ALL)**

```sql
SELECT
  City
FROM
  /shared/examples/ds_inventory/suppliers
INTERSECT ALL
SELECT
  City
FROM
  /shared/examples/ds_orders/customers
```

**LEFT OUTER JOIN**

**Example of LEFT OUTER JOIN**

```sql
SELECT *
FROM
  /shared/examples/ds_orders/products products
LEFT OUTER JOIN
  /shared/examples/ds_orders/orderdetails orderdetails
ON
  products.ProductID = orderdetails.ProductID
```

**ORDER BY**

**Example of ORDER BY**

If the sort order is unspecified, this function sorts columns in ascending order, as ascending is the default sorting order. If you want the columns to be sorted in the descending order, you need to specify it as in the example given below.

Note that unless an ORDER BY is specified, the order is undefined. In fact, the sort order can be different with two runs of the same identical SQL ORDER BY, if specified, is ascending by default.
You can include this function as you would in the SELECT clause.

**Example of ORDER BY without a function**

```sql
SELECT *
FROM /shared/examples/ds_inventory/inventorytransactions
ORDER BY ProductID, UnitsSold DESC
```

Meaning: Select all columns from the Inventory Transactions table and sort by the fields Product ID (in ascending order) and UnitsSold (in descending order).

**Example of ORDER BY with a multiplication function**

```sql
SELECT "Inventory Transactions".ProductId, ("Inventory Transactions".UnitsSold) * ("Inventory Transactions".UnitPrice)
FROM composite.user.ds_1_access@"Inventory Transactions"
ORDER BY ProductID, (UnitsSold) * (UnitPrice)) DESC
```

Meaning: Select all columns from the Inventory Transactions table and sort them by ProductID in the ascending order, and the value obtained by multiplying the value of UnitsSold by UnitPrice in the descending order.
**RIGHT OUTER JOIN**

**Example of RIGHT OUTER JOIN**

```sql
SELECT *
FROM /shared/examples/ds_orders/products products
RIGHT OUTER JOIN /shared/examples/ds_orders/orderdetails orderdetails
ON products.ProductID = orderdetails.ProductID
```

**SELECT**

Besides other features of a regular SQL SELECT statement, Composite supports the definition of "virtual columns" in the projection list for a view. Once virtual columns are declared, you can use them in a query anywhere a literal can be used.

The prime use of a virtual column is in procedures included in the FROM clause of a query. However, you can use virtual columns also in the WHERE, HAVING and JOIN ON clauses. Including them in the GROUP BY and ORDER BY clauses is legal, but it has no effect (like literals).

**Syntax**

A virtual column is defined using the following syntax:

```
{DECLARE columnName columnType}
```

OR

```
{DECLARE columnName columnType DEFAULT literalValue}
```

The virtual column is declared in the SELECT clause, as follows:

```
SELECT c1, {DECLARE someColumn VARCHAR(40)},
c2, {DECLARE otherColumn INTEGER DEFAULT 40} ...
```

**Example**

A view, V1, is defined by the following SELECT:
SELECT
T1.column1, {DECLARE columnName INTEGER DEFAULT 50},
T1.column2
FROM
/some/table T1, ProcedureOne(5, columnName) P1,
ProcedureTwo(concat(columnName,'abc')) P2
WHERE
(columnName > T1.column1) AND (T1.someKey = P2.someKey)

Notes

- Virtual columns are unqualified, so their names must be unique and different from the names of items in the FROM clause.
  For example, if you select FROM a table with a column named ColumnOne, your virtual column should not be named ColumnOne in order to avoid the confusion whether the table's column or the virtual column should be used.
- When a query using virtual columns is executed, the query engine analyzes the predicates (such as a WHERE clause) to look for columnName = literal expressions. These clauses are removed from the query and the literal is replaced, much like a ? (question mark) is replaced in a prepared statement.
  For example, the following statement:
  
  ```
  SELECT * FROM V1 WHERE columnName = 99
  ```
  would become:
  
  ```
  SELECT
  T1.column1, 99, T1.column2
  FROM
  /some/table T1, ProcedureOne(5,99) P1,
  ProcedureTwo(concat(99,'abc')) P2
  WHERE
  (99 > T1.column1) AND (T1.someKey = P2.someKey)
  ```

- The use of columnName = literal is important. Other types of comparison operators do not result in setting the value. The literal can be a single literal or an expression containing only functions and literals, like `concat('abc','def')`. 
The relationship optimization applies to virtual columns. This means that if the query has `columnName = otherColumn` and there is a predicate for `otherColumn = 5`, the query engine will figure out that `columnName = 5` is also true and set that for you.

It is possible when using outer joins for the `WHERE` clause to not be legally applied to the inner side of the join. When this happens, the query engine will not be able to do the replacement and you will get an error that may or may not be clear.

If no `DEFAULT` value is specified for a virtual column, the column’s value must be specified in the `WHERE` clause or you will get an error.

If a `DEFAULT` value is specified it will be used if no `WHERE` clause setting is found.

If a virtual column is set to more than one value, you will get an error.
SEMI-JOIN (to a procedure)

SEMI-JOIN to a procedure is the logical equivalent of a semi-join to a table.

Syntax

```
(table expression)
[LEFT OUTER|RIGHT OUTER|INNER] PROCEDURE JOIN
(procedure) ProcedureAlias
ON (condition expression)
```

This syntax conveys that for each unique-value set of procedure inputs, the procedure on the right will be called once. The results from each call will be effectively "UNION ALL'd" together and treated as a row set to be fed into the join. The join operates just like a non-procedure join of the same type.

Example

```
(T1 LEFT OUTER JOIN T2 ON T1.x = T2.x)
INNER PROCEDURE JOIN
MyProc(T1.y+T2.y) P1 ON (T1.z = P1.z)
```

Remarks

- The special syntax given here always has a procedure on the right side and allows you to deviate from the normal rule that a procedure’s input parameters must be literal expressions.
- When using this syntax, the procedure’s input parameters may include references to any item from the table expression on the left, and only from that context. That is, only values from inside the left-side subquery can be used. The values from other scopes cannot be used.
- All the input value combinations are tracked and are not repeated to call the procedure again.
- On using the `PROCEDURE` keyword. Without the `PROCEDURE` keyword, your procedure is called exactly once. With the keyword, your procedure is called zero or more times depending on the left side of the join.
UNION

UNION works like UNION ALL (page 133) except it does not produce duplicate rows.

Syntax

```
<query-expression>
UNION (ALL)
<query-expression>
```

Example of UNION

The following sample query lists the states where authors and publishers are located in
the authors table and publishers table respectively.

```
SELECT state FROM authors
UNION
SELECT state FROM publishers
```

Notes

- The SELECT clause lists in the two queries must have the same number of
  projections.
- Corresponding columns in the two queries must be listed in the same order in
  the two queries.
- Corresponding columns must have the same data type or must be implicitly
  convertible to the same type.
- An ORDER BY clause can appear in only the final query of the union statement.
  The sort is applied to the final combined result.
- GROUP BY and HAVING can be specified in the individual queries only. They
  cannot be used to affect the final result.
- For the purposes of a Set operation, two NULLS are duplicates of each other.
UNION ALL

UNION ALL combines two tables, row by row. Implement UNION ALL by manual the SQL panel of Studio Modeler

Multiple column selections may be made, but the number of columns and the column data types should match. All queries in an SQL statement containing the UNION ALL function must have an equal number of expressions in their target lists, as shown in the following example.

```sql
SELECT ProductID, ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products
UNION ALL
SELECT ProductID, ProductName, UnitPrice
FROM /shared/examples/ds_inventory/products products_1
```

**Example of UNION ALL**

Suppose that table T1 has columns, C1, C2, and C3, and table T2 has columns, C1i, C1ii, C1iii.

If T1 looked like this:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Hello</td>
<td>Goodbye</td>
</tr>
<tr>
<td>002</td>
<td>Hola</td>
<td>Adios</td>
</tr>
<tr>
<td>003</td>
<td>Aloha</td>
<td>Aloha</td>
</tr>
</tbody>
</table>

And T2 had values like this:

<table>
<thead>
<tr>
<th>C1i</th>
<th>C1ii</th>
<th>C1iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>003</td>
<td>Aloha</td>
<td>Aloha</td>
</tr>
</tbody>
</table>

133
Then the following SQL would yield the table that follows it:

```
SELECT C1 C2 C3 FROM T1
UNION ALL
SELECT Ci Cii Ciii FROM T2
```

This result set from UNION ALL contrasts with the output of the UNION function which would omit the repeated value of 003.

**UPDATE**

You can update a physical table, view based on a single physical table. See "INSERT/UPDATE/DELETE on Views" on page 136 for rules on updating views.

**Syntax**

```
UPDATE <table>
SET <column> = <expression>, <column> = <expression>*
[WHERE <criteria>]
```

**Remarks**

- The **WHERE** clause in a **DELETE** statement is optional. The rules for the **WHERE** clause of an **UPDATE** statement is the same as the rules for the **WHERE** clause of a **SELECT** statement.
Sub queries in the SET clause are not permitted.
For example, UPDATE <T> SET x = (SELECT y FROM T1)
If a non-nullable column is set to null, then the data source layer will throw a runtime exception.
If the column is set to an invalid value, then the data source layer will throw an runtime exception

Examples of UPDATE
The following examples illustrate the use of UPDATE.

Using UPDATE with SET

PROCEDURE sc5()
BEGIN
  UPDATE
    /shared/examples/ds_inventory/products
  SET
    ProductName = 'Apple';
END

Using UPDATE with SET and WHERE

PROCEDURE sc6()
BEGIN
  UPDATE
    /shared/examples/ds_inventory/products
  SET
    ProductName = 'Lexington Z24'
  WHERE
    ProductID = 5;
END
Using UPDATE and a subquery

PROCEDURE sc8()
BEGIN
  UPDATE /shared/examples/ds_orders2/products
  SET
  ProductName = 'abc'
  WHERE
  ProductID IN
  (SELECT ProductID FROM
   /shared/examples/ds_orders2/orderdetails);
END

INSERT/UPDATE/DELETE on Views

INSERT/UPDATE/DELETE on views are supported as defined by SQL standard.

The following statement summarizes Composite’s support for INSERT/UPDATE/DELETE on views:

A view is updatable only if it is defined to be a direct row and column subset of some base table or a direct row and column subset of some other updatable view.

An updatable view is one that follows all the following conditions:

- The SQL of the view cannot include DISTINCT, GROUP BY, or HAVING
- The FROM clause of the view refers to exactly one table reference, and that table reference identifies either a base table or an updatable view.
- Derived columns are not updatable.
- A view with an aggregate expression in projection is not updatable irrespective of whether GROUP BY is present or not.

WHERE

Example of WHERE

SELECT
  ProductID, ProductName, ProductDescription
FROM /shared/examples/ds_inventory/products Products
WHERE ReorderLevel > 5

WITH

The WITH clause starts an SQL query, defining the aggregations, which in turn can be referred in the main query and in other WITH statements as if they are real tables.

Syntax

WITH
queryName AS (query expression)
[ , ...]
mainQueryExpression

WITH clauses may also refer to a sibling WITH definition

WITH
X as (SELECT * From Foo),
Y as (SELECT * From X)
Select * From Y

Releases prior to 4.5 did not allow the WITH element from 'Y' to refer to the WITH element 'X'.

Remarks

- You can first name a query expression and use it within the main query expression by referring to it. If an expression occurs more than once or is complex, moving it out provides clarity.
- The WITH query is run once and the results are stored in the equivalent of a temporary table, which is scanned whenever the results are used. For certain types of queries, this scanning can reduce the burden on the data source.
Subqueries

Example
Suppose that you have a Web service that returns employee data with the following columns:
- employeeNo (the number of employee)
- employeeName (the name of employee)
- manager (the employee number of the employee’s manager)

The following query lists all the employees with the details on their respective managers:

```
WITH us_employees AS
    (SELECT employeeNo, employeeName, manager FROM employee_webservice WHERE country = 'US')
SELECT e.employeeNo, e.employeeName, 'works for', e.manager, 'who is', m.employeeNo, m.employeeName
FROM us_employees e, us_employees m
WHERE e.manager = m.employeeNo
```

The advantage of using WITH in this scenario is that it will invoke the Web service only once, which in turn will enhance query execution performance.

Note: Differences in the output from different release versions may occur in the unusual case where multiple WITH statements define the same variable name with different SQL within the same procedure. Previous release versions used the first name definition and now the last name definition takes priority.

Subqueries
You can embed an SQL SELECT statement within another. When an SQL statement is embedded within another, it is referred to as a subquery. Composite supports using subqueries as values. See the section “EXISTS” on page 106.
Scalar Subqueries and Correlated Subqueries

Two types of subqueries are recognized: scalar subqueries and correlated subqueries.

Scalar Subqueries

A scalar subquery is a subquery that returns a single value. It can be used anywhere a single column value or literal is legal.

Subqueries can reside within the WHERE clause, the FROM clause, or the SELECT clause.

Example

```
SELECT *
FROM table1
WHERE column1 = (SELECT column1 FROM table2);
```

Correlated Subqueries

A correlated subquery is a subquery that contains a reference to a table that also appears in the outer query. In the following Syntax section, the correlated subquery is rendered in blue.

Syntax

```
SELECT outer_column
FROM outer_table
WHERE outer_column_value IN
  (SELECT inner_column FROM inner_table
   WHERE inner_column = outer_column)
```

Remarks
Notice in the syntax given above the reference to the outer query from inner query `outer_column`. This reference is called the `correlation variable`.

A correlated subquery is used if a statement needs to process a table in the inner query for each row in the outer query.

A correlated subquery cannot be evaluated independent of its outer query. The inner query is dependant on the data from the outer query.

Correlated subquery differs from simple query in its order of execution and in the number of times it is executed. A correlated subquery is executed repeatedly - once for each candidate row selected by the outer query. It always refers to the table mentioned in the `FROM` clause of the outer query.

**Example**

```sql
SELECT
  name
FROM
  salesreps mgrs
WHERE
  age > 40 AND mgrs.EMP_NO IN
  (SELECT manager
    FROM salesreps emps
    WHERE emps.quota > emps.sales
    AND emps.rep_office <> mgrs.rep_office)
```

The above query lists the managers who are over 40 and who manage a sales person who is over quota and who does not work in the same sales office as the manager.

**Consolidated List of Reserved Words**

The following table lists the reserved words in Composite.

- Note that some of these reserved words are not valid keywords.
- Do not use any reserved word as an identifier.
- If you do choose to use a reserved word as an identifier, enclose it in double quotes.
### Table 47. Reserved Words in Composite

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs absolute acos action add all</td>
<td>begin between bigint binary bit bit_length</td>
<td>begin between bigint binary bit bit_length</td>
<td>begin between bigint binary bit bit_length</td>
</tr>
<tr>
<td>allocate alter and any are as</td>
<td>boolean boolean_condition both breadth by</td>
<td>boolean boolean_condition both breadth by</td>
<td>boolean boolean_condition both breadth by</td>
</tr>
<tr>
<td>asc asin assertion asterisk at atan</td>
<td>call cascade cascaded case case_expression case_expression</td>
<td>call cascade cascaded case case_expression case_expression</td>
<td>call cascade cascaded case case_expression case_expression</td>
</tr>
<tr>
<td>authorization avg</td>
<td>cast cast_function catalog ceiling char</td>
<td>cast cast_function catalog ceiling char</td>
<td>cast cast_function catalog ceiling char</td>
</tr>
<tr>
<td>char_length character character_length</td>
<td>check close coalesce</td>
<td>check close coalesce</td>
<td>check close coalesce</td>
</tr>
<tr>
<td>collate collation column comma commit concat</td>
<td>connect connection constant constraint constraints continue</td>
<td>connect connection constant constraint constraints continue</td>
<td>connect connection constant constraint constraints continue</td>
</tr>
<tr>
<td>convert corresponding cos cot count count_star</td>
<td>create current current_date current_time current_timestamp</td>
<td>create current current_date current_time current_timestamp</td>
<td>create current current_date current_time current_timestamp</td>
</tr>
<tr>
<td>current_user cursor cycle</td>
<td>date date_function day deallocate dec decimal</td>
<td>date date_function day deallocate dec decimal</td>
<td>date date_function day deallocate dec decimal</td>
</tr>
<tr>
<td>declare declared_variable default deferrable deferred degrees</td>
<td>declare declared_variable default deferrable deferred degrees</td>
<td>declare declared_variable default deferrable deferred degrees</td>
<td>declare declared_variable default deferrable deferred degrees</td>
</tr>
</tbody>
</table>
Table 47. Reserved Words in Composite

<table>
<thead>
<tr>
<th>Reserved Words in Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete delimited_identifier</td>
</tr>
<tr>
<td>diagnostics disconnect</td>
</tr>
<tr>
<td>domain</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>except</td>
</tr>
<tr>
<td>exponent</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>foreign</td>
</tr>
<tr>
<td>function</td>
</tr>
<tr>
<td>ge</td>
</tr>
<tr>
<td>group</td>
</tr>
<tr>
<td>having</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>independent</td>
</tr>
<tr>
<td>input insensitive</td>
</tr>
<tr>
<td>interval</td>
</tr>
</tbody>
</table>
### Table 47. Reserved Words in Composite

<table>
<thead>
<tr>
<th>Reserved Words in Composite</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>J</strong></td>
</tr>
<tr>
<td>join</td>
</tr>
<tr>
<td>joined_table</td>
</tr>
<tr>
<td><strong>K</strong></td>
</tr>
<tr>
<td>key</td>
</tr>
<tr>
<td><strong>L</strong></td>
</tr>
<tr>
<td>language</td>
</tr>
<tr>
<td>last</td>
</tr>
<tr>
<td>le</td>
</tr>
<tr>
<td>leading</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>left</td>
</tr>
<tr>
<td>left_brace</td>
</tr>
<tr>
<td>left_bracket</td>
</tr>
<tr>
<td>left_paren</td>
</tr>
<tr>
<td>length</td>
</tr>
<tr>
<td>level</td>
</tr>
<tr>
<td>like</td>
</tr>
<tr>
<td>local</td>
</tr>
<tr>
<td>log</td>
</tr>
<tr>
<td>longvarchar</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>lower</td>
</tr>
<tr>
<td>lt</td>
</tr>
<tr>
<td><strong>M</strong></td>
</tr>
<tr>
<td>match</td>
</tr>
<tr>
<td>max</td>
</tr>
<tr>
<td>min</td>
</tr>
<tr>
<td>minus</td>
</tr>
<tr>
<td>minute</td>
</tr>
<tr>
<td>module</td>
</tr>
<tr>
<td>modulo</td>
</tr>
<tr>
<td>month</td>
</tr>
<tr>
<td>multiply</td>
</tr>
<tr>
<td><strong>N</strong></td>
</tr>
<tr>
<td>names</td>
</tr>
<tr>
<td>national</td>
</tr>
<tr>
<td>natural</td>
</tr>
<tr>
<td>nchar</td>
</tr>
<tr>
<td>next</td>
</tr>
<tr>
<td>no</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>not_eq</td>
</tr>
<tr>
<td>null</td>
</tr>
<tr>
<td>null_if</td>
</tr>
<tr>
<td>number</td>
</tr>
<tr>
<td>number_decimal</td>
</tr>
<tr>
<td>number_float</td>
</tr>
<tr>
<td>numeric</td>
</tr>
<tr>
<td><strong>O</strong></td>
</tr>
<tr>
<td>octet_length</td>
</tr>
<tr>
<td>of</td>
</tr>
<tr>
<td>on</td>
</tr>
<tr>
<td>only</td>
</tr>
<tr>
<td>open</td>
</tr>
<tr>
<td>option</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>order</td>
</tr>
<tr>
<td>order_clause</td>
</tr>
<tr>
<td>out</td>
</tr>
<tr>
<td>outer</td>
</tr>
<tr>
<td>output</td>
</tr>
<tr>
<td>overlaps</td>
</tr>
<tr>
<td><strong>P</strong></td>
</tr>
<tr>
<td>pad</td>
</tr>
<tr>
<td>partial</td>
</tr>
<tr>
<td>path</td>
</tr>
<tr>
<td>pi</td>
</tr>
<tr>
<td>pipe</td>
</tr>
<tr>
<td>plus</td>
</tr>
<tr>
<td>position</td>
</tr>
<tr>
<td>position_function</td>
</tr>
<tr>
<td>power</td>
</tr>
<tr>
<td>precision</td>
</tr>
<tr>
<td>prepare</td>
</tr>
<tr>
<td>preserve</td>
</tr>
<tr>
<td>Reserved Words in Composite</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>primary</td>
</tr>
<tr>
<td>Q</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>relative</td>
</tr>
<tr>
<td>right_brace</td>
</tr>
<tr>
<td>rows</td>
</tr>
<tr>
<td>select</td>
</tr>
<tr>
<td>session</td>
</tr>
<tr>
<td>slash_expression</td>
</tr>
<tr>
<td>space</td>
</tr>
<tr>
<td>sqlstate</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>to_char</td>
</tr>
</tbody>
</table>
### Table 47. Reserved Words in Composite

<table>
<thead>
<tr>
<th>Reserved Words in Composite</th>
<th>translate</th>
<th>translation</th>
<th>trim</th>
<th>true</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unary_minus</td>
<td>unary_plus</td>
<td>union</td>
<td>unique</td>
<td>unknown</td>
<td>unsigned_integer</td>
</tr>
<tr>
<td>until</td>
<td>update</td>
<td>upper</td>
<td>usage</td>
<td>use</td>
<td>user</td>
</tr>
<tr>
<td>user_function</td>
<td>using</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>values</td>
<td>varbinary</td>
<td>varchar</td>
<td>varying</td>
<td>vector</td>
</tr>
<tr>
<td>vertbar</td>
<td>view</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>when</td>
<td>whenever</td>
<td>where</td>
<td>where_condition</td>
<td>while</td>
<td>with</td>
</tr>
<tr>
<td>work</td>
<td>wrapped</td>
<td>write</td>
<td>ws</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>year</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2

Composite SQL Script

The Composite SQL Script language is similar to the stored procedure languages offered by relational database systems (RDBMS) and serves the same purpose. The script's mechanism is to allow logic to be performed in the server.

This chapter provides complete reference to the SQL Script language with several basic examples.

Note that this chapter does not provide advanced-level programming tutorials.

Topics for the SQL Script language reference:
- “Language Concepts” on page 147
- “Procedures and Structure” on page 163
- “Statement Reference” on page 176
- “Examples” on page 222

SQL Script Overview

A SQL script is a procedure that employs procedure declaration, parameters, statements, variables, data types, procedure calls, SQL keywords, dynamic SQL, conditionals, loops, cursors (simple and streaming), exceptions, and transactions. The keywords for these various components are listed here.

Procedure Declaration and Parameters

PROCEDURE; IN, INOUT, OUT

Compound Statement

BEGIN/END

Variables

DECLARE, SET, DEFAULT
Language Concepts

The following sections cover the basic concepts of the SQL Script language.

- “Identifiers” on page 148
- “Data Types” on page 149
- “Literal Values” on page 156
- “Variables” on page 156
- “Attributes” on page 159
- “Expressions” on page 154
- “Keywords” on page 162

Data Types
- DECLARE TYPE, BOOLEAN, ROW, XML

Procedure Call
- CALL

Path to a Resource
- PATH

SQL Keywords
- SELECT INTO, INSERT, UPDATE, DELETE

Dynamic SQL
- EXECUTE IMMEDIATE

Conditionals
- IF/THEN/ELSE, CASE/WHEN

Loops
- LOOP, WHILE, REPEAT/UNTIL, FOR, ITERATE, LEAVE

Cursors
- ROW, CURSOR, OPEN, CLOSE, FETCH, SELECT, PIPE (for streaming)

Exceptions and Transactions
- RAISE, EXCEPTION, CURRENT_EXCEPTION (for exceptions)
- TRANSACTION, INDEPENDENT, COMMIT, ROLLBACK (for transactions)
Identifiers

An identifier is a user-defined sequence of one or more characters forming a unique name.

- Identifiers should begin with an alphabetical character, and can contain only alpha-numeric characters, _ (underscore), $ (dollar sign), or # (hash symbol).
- Identifiers are used for the following:
  - procedure name, parameter name, cursor name, field name, variable name, cursor variable name, data type name, exception name, and label for a block (such as, BEGIN/END, LOOP, WHILE, REPEAT, FOR, LEAVE, ITERATE)
- Composite SQL Script resolves identifiers by a set of processing rules.

Identifiers within SQL expressions are first evaluated by looking locally in the SQL context and then outer scopes.

Identifier resolution is first attempted within the local SQL context. If an identifier is resolved within the local SQL context then the SQL engine will not continue processing and searching for additional matching identifiers in other scopes. Note that the SQL context space is never case-sensitive, so differences in capitalization will not distinguish names that match an identifier within the SQL context.

SQL Script identifier resolution proceeds from local to parent contexts using the smallest prefix basis from the current scope outward to schema-level scopes.

Thus identifier name matches in database columns in the SQL WHERE clause take precedence over the names of local variables, procedure names, or formal parameters, as searching for identifier matches stops if a locally meaningful symbol hit is found.

SQL Script also resolves identifiers with symbols or other characters without regard to case sensitivity. If the exact identifier is not present in the SQL context, then the local variable space is evaluated, and then outward until an identifier match is found.

Of course, if no matches are found an undeclared identifier error is given.
Identifiers cannot be one of the SQL Script keywords (see “Keywords” on page 162), unless the keyword is escaped using double quotes.

Example declarations of variables that are SQL Script keywords:

```sql
DECLARE "VALUE" INTEGER;
DECLARE "CURSOR" CURSOR;
```

Note that here the SQL Script keywords VALUE and CURSOR are enclosed within double quotes.

Escaping an identifier with double quotes can also allow it to contain characters that would otherwise not be legal, such as spaces, dashes, or characters from other languages.

Example declarations of variables that contain otherwise illegal characters:

```sql
DECLARE "First Name" VARCHAR(40);
DECLARE "% Returned" DOUBLE;
```

**Characters**

Alpha-numeric characters, separators, and special characters are used.

Valid alphabetical characters: a-z, A-Z

Valid numerical characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Valid separators: , (comma), ; (semicolon), ' (single quotes)

Valid special characters: _ (underscore), / (forward slash), $ (dollar sign), # (hash symbol)

**Data Types**

SQL-supported character strings, numeric, date, time, and datetime data types are supported in SQL Script.

Additionally, other data types—BLOB, CLOB, ROW and XML—are supported.

SQL Script allows the declaration of custom data types for convenience and clarity. Custom types may be declared locally or made PUBLIC. See “DECLARE TYPE” on page 191 for details on how to declare custom types.
Once declared, a custom type’s name may be used anywhere one of the built-in types would be used.

Example:

```sql
DECLARE TYPE SocialSecurityType VARCHAR(12);
DECLARE ssn SocialSecurityType;
DECLARE data ROW(name VARCHAR(40), ssn SocialSecurityType);
```

A custom type may also be made public (see “DECLARE TYPE” on page 191).

A public type in another procedure may be accessed by specifying the fully qualified path to that procedure, using a period, then the name of the type.

If the declaration in the preceding example is in a procedure named TypeSample in the folder `/shared/examples`, the type can be referenced as follows:

```sql
DECLARE ssn /shared/examples/TypeSample.SocialSecurityType;
```

Fully qualified public type references are legal anywhere a type can be used.

The following table lists all the data types supported. All types with optional sizes have default values, as noted.

### Table 48. SQL Script Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Integral Numeric Types</strong></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>0 or 1</td>
</tr>
<tr>
<td>TINYINT</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-(2^31) to +(2^31 - 1)</td>
</tr>
<tr>
<td>INT</td>
<td>alias for INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>-(2^63) to +(2^63 - 1)</td>
</tr>
<tr>
<td><strong>Non-Integral Numeric Types</strong></td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td>approximately, 7-digit precision floating point.</td>
</tr>
</tbody>
</table>
### Table 48. SQL Script Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>Alias for FLOAT</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>approximately, 17-digit precision floating point.</td>
</tr>
<tr>
<td>DECIMAL[n, m]</td>
<td>Fixed precision number with up to &quot;n&quot; digits total and up to &quot;m&quot; digits to the right of the decimal. Default: (32, 2)</td>
</tr>
<tr>
<td>NUMERIC[n, m]</td>
<td>Same as DECIMAL, except defaults to (32, 0).</td>
</tr>
<tr>
<td>DATE</td>
<td>Date and Time Types</td>
</tr>
<tr>
<td>TIME</td>
<td>Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Date and Time Types</td>
</tr>
<tr>
<td>CHAR[n]</td>
<td>String and Binary Types</td>
</tr>
<tr>
<td>VARCHAR[n]</td>
<td>Character string of exactly &quot;n&quot; characters padded with spaces. Default: (255)</td>
</tr>
<tr>
<td>BINARY[n]</td>
<td>Binary string exactly &quot;n&quot; bytes padded with zero bytes. Default: (255)</td>
</tr>
<tr>
<td>Data Type</td>
<td>Range/Value</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>Binary string of up to “n” bytes without padding.</td>
</tr>
<tr>
<td></td>
<td>Default: (255)</td>
</tr>
<tr>
<td>Other Types</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>A value of TRUE or FALSE. BOOLEAN is not a legal parameter type.</td>
</tr>
<tr>
<td>CURSOR</td>
<td>Consists of a set of fields, also called columns. If no list of fields is</td>
</tr>
<tr>
<td></td>
<td>provided, the CURSOR is untyped.</td>
</tr>
<tr>
<td>CURSOR(...)</td>
<td>A CURSOR can also be declare by referencing a ROW Type instead of specifying</td>
</tr>
<tr>
<td></td>
<td>fields directly.</td>
</tr>
<tr>
<td>CURSOR(rowType)</td>
<td></td>
</tr>
<tr>
<td>ROW(...)</td>
<td>Consists of a set of fields, also called columns.</td>
</tr>
</tbody>
</table>
Type Modifier

A type-modifier, named PIPE, is used for streaming, and it is used only in procedure parameter declarations to pipeline the output. For details, see “PIPE - Type Modifier” on page 166.
Expressions

There are two types of expressions: Value Expressions and Conditional Expressions.

Value Expressions

A value expression is anything that resolves to a value.

Syntax

The syntax for a value expression is identical to a projection in a `SELECT` statement, except that instead of using column names you can use variable names in a value expression.

Remarks

- Cursor variables cannot be used in a value expression by themselves, although attributes of cursor variables can be used. See “DECLARE CURSOR” on page 184 for information on declaring cursor variables, and “Attributes of Cursors” on page 159 for information on cursor attributes.
- The keyword `CURRENT_EXCEPTION` cannot be used in a value expression by itself, although attributes of it can be used. See “Attributes of CURRENT_EXCEPTION” on page 161 for information on using `CURRENT_EXCEPTION`.

Errors

The following table describes the errors that may occur while resolving a value expression.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined variable</td>
<td>An identifier is encountered that is not defined in the current scope.</td>
</tr>
<tr>
<td>Incorrect use of a cursor</td>
<td>A cursor is used in a value expression.</td>
</tr>
</tbody>
</table>
Conditional Expressions

A conditional expression is anything that resolves to a boolean.

Syntax

The syntax for a conditional expression is identical to what you can use as a WHERE clause, except that instead of using column names you can use variable names in a conditional expression.

Remarks

- Cursor variables can be used in a conditional expression only with the keyword IS NULL or IS NOT NULL. Cursor variables cannot be used in other conditional expressions, although attributes of cursor variables can be used. See “DECLARE CURSOR” on page 184 for information on declaring cursor variables, and “Attributes of Cursors” on page 159 for information on cursor attributes.

- A boolean variable or literal can be used as a condition. See “Literal Values” on page 156 for information on declaring literals.

- The keyword CURRENT_exception cannot be used in a conditional expression by itself, although attributes of it can be used. See “Attributes of CURRENT_EXCEPTION” on page 161 for information on using CURRENT_EXCEPTION.
Errors
The following table describes the errors that may occur while resolving a conditional expression.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined variable</td>
<td>An identifier is encountered that is not defined in the current scope.</td>
</tr>
<tr>
<td>Incorrect use of a cursor</td>
<td>A cursor is used in a conditional expression with something other than IS NULL or IS NOT NULL.</td>
</tr>
<tr>
<td>Incorrect use of CURRENT_EXCEPTION</td>
<td>The keyword CURRENT_EXCEPTION is used in a conditional expression.</td>
</tr>
</tbody>
</table>

Literal Values
All of the literal values legal in SQL are legal in SQL Script. In addition, two new types—ROW and XML—are available and they need to be defined.

The symbols TRUE and FALSE are now reserved for use as literal BOOLEAN values.

Literal values are delimited by single quotes ('string'). To specify an apostrophe within a string, put two apostrophes in a row (' ').

Syntax
ROW( <valueExpression>, ... )
There is no literal format for an XML type. Use the following syntax to create an XML type.
CAST('xml string' AS XML)
There is no literal format for a cursor type. See "DECLARE CURSOR" on page 184 for defining cursors.

Variables
Variables have scoping rules.
Scoping Rules
You can define a non-cursor variable by specifying its name and data type, and initializing it with a default value. See “DECLARE CURSOR” on page 184 for defining cursor variables.

Syntax
DECLARE <varName>[,...] <dataType>
[DEFAULT <valueExpression>]
The DEFAULT syntax is optional and is used to initialize a variable.

Remarks
- A variable can be declared within a block that has the same name as a variable in a parent block. Parameters are treated as if they were defined in the main block of the procedure.
- String type variables are delimited by single quotes (’string’). To specify an apostrophe within a string, put two apostrophes in a row (’ ’).
- You can declare variables, parameters, and column definitions that are of type BLOB or CLOB.
- You can declare more than one variable at a time, provided all the variables are of the same data type but each one has a unique name.
- The value expression, indicated by <valueExpression> in the syntax, can use IN parameters, previously declared variables in this block, and any variables in parent blocks. In the current block, the value expression cannot use variables that are defined later. If the value expression’s type does not match the variable’s type, an implicit cast will be performed (if possible). See “Procedure Header” on page 165 for information on IN parameters.
- Any variable that is not initialized with a DEFAULT clause has the value NULL.
- If the evaluation of the value expression causes an exception, any other declared variables that have not yet been initialized are set to NULL before entering the exception handler.

You can define a new cursor variable by providing a unique name and optionally specifying its data type. See “DECLARE CURSOR” on page 184, “Attributes of
Cursors” on page 159, “OPEN” on page 213, “FETCH” on page 204, and “CLOSE” on page 182 for additional information on cursors.

**Syntax**

```sql
DECLARE <varName> CURSOR
[<dataType>]
```

The data type, indicated by `<dataType>` in the Syntax, is optional and can be a named ROW data type or the syntax for a ROW data type. The syntax for a ROW data type is: `<colName> <dataType> [,…]`. There are no attributes on a ROW variable.

You access a row using `rowVar.columnName` to get a column.

**Remarks**

- When declared, the cursor variable is initialized to `NULL`. It cannot be initialized to any other value at declaration.
- A cursor variable with a type can be assigned from any cursor with the same ROW type, or to any cursor variable with exactly the same ROW type.
- A cursor variable without a type can be assigned from any cursor, or to any cursor. Assigning to a typed cursor forces a runtime schema match comparison and raises an exception on a mismatch.
- Assigning a cursor creates a reference to the original cursor’s state. This means that opening, closing, or fetching from the original cursor or the variable has the same effect and alters what the other would see. See “OPEN” on page 213, “CLOSE” on page 182, and “FETCH” on page 204 for details on opening, closing, and fetching actions on cursors.

**Using Variables**

Variables may be used in SQL Script expressions anywhere a literal value would be legal. For example, both `1 + 1` and `x + y` are legal expressions (assuming `'x'` and `'y'` are declared variables).
Examples

```sql
PROCEDURE p ( )
BEGIN
  DECLARE a INTEGER;
  DECLARE b DATE;
  DECLARE c TIME;
  DECLARE d TIMESTAMP;
  DECLARE e DECIMAL;
  DECLARE f FLOAT;
  DECLARE g VARCHAR;
  DECLARE h CHAR;
END

PROCEDURE p ( )
BEGIN
  DECLARE x INTEGER;

  SET x = 1;
  DECLARE x INTEGER; -- illegal
END
```

Attributes

You can obtain the attributes of cursors and current exceptions, as described in this section.

Attributes of Cursors

You can obtain the attributes of a cursor. See “DECLARE CURSOR” on page 184, “OPEN” on page 213, “FETCH” on page 204, and “CLOSE” on page 182 for other details on cursors.

Syntax

```sql
<cursor>.<attribute>
```
The following table describes cursor attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISOPEN</td>
<td>A boolean that indicates whether the cursor is open or not.</td>
</tr>
<tr>
<td>ROWTYPE</td>
<td>The row data type for the cursor. NULL, for an untyped cursor.</td>
</tr>
<tr>
<td>ROWCOUNT</td>
<td>Number of rows fetched from the cursor if open. NULL, if not open.</td>
</tr>
<tr>
<td>FOUND</td>
<td>A boolean that is true if the last fetch from the cursor found a row. NULL, if not open or open and not fetched from.</td>
</tr>
</tbody>
</table>

**Example**

```sql
-- Returns the n-th value of a cursor of VARCHARs
PROCEDURE nth(IN n INTEGER, IN crs CURSOR(name VARCHAR), OUT name VARCHAR)
   a_lab:
   BEGIN
      IF NOT crs.ISOPEN THEN
         OPEN crs;
      END IF;
      LOOP
         FETCH crs INTO name;
         IF NOT crs.FOUND OR nth >= crs.ROWCOUNT THEN
            LEAVE a_lab;
         END IF;
      END LOOP;
      CLOSE crs;
   END
```
Attributes of CURRENT_EXCEPTION

You can obtain the attributes of an exception while within the exception handler. See also “DECLARE EXCEPTION - public” on page 189, “External Exceptions” on page 175, and “Raising and Handling Exceptions” on page 174.

Syntax

CURRENT_EXCEPTION.<attribute>

The following table describes exception attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>
| NAME      | A string that is the exception’s name. This name will be fully qualified, as follows: 
/na1/na2/procedure.s1.s2.exceptionName 
The na1 and na2 are namespace elements of the path. The s1 and s2 are compound statement blocks and are either named according to the label on that block or as unnamed# where # is an integer value. |
| ID        | An integer that is the exception’s system ID. All user exceptions have the ID -1 (negative one). System exceptions all have unique IDs. |
| MESSAGE   | The VARCHAR(255) value defined for the current exception. If no value is defined for the exception, then this attribute is NULL. |
| TRACE     | The VARCHAR(2048) value defined that includes the exception stack trace as a string. |

Remark

- If the exception handler includes a compound statement, CURRENT_EXCEPTION within the BEGIN portion refers to the current exception of the parent scope, but within the exception handler portion of the sub-scope CURRENT_EXCEPTION
refers to the local exception and there is no way to access the parent exception. See “Compound Statement” on page 167 for details on compound statements.

**Example**

```sql
PROCEDURE p (IN x INTEGER, OUT result VARCHAR)
BEGIN
    CALL /shared/f(x);
    EXCEPTION
    ELSE
        IF CURRENT_EXCEPTION.MESSAGE IS NOT NULL THEN
            SET result = CURRENT_EXCEPTION.MESSAGE;
        ELSE
            SET result = CURRENT_EXCEPTION.NAME;
        END
    END
    MESSAGE: 'x must be > 0. x = -123'
    NAME: '/shared/f.illegal_arg_ex'
```

**Keywords**

SQL Script keywords are not case-sensitive. This document uses upper case letters to render keywords in order to separate them from other words.

Note that you can use these keywords in a SQL Script as long as they are enclosed within double quotes, as for example:

```sql
SELECT "BEGIN" INTO ...
```

The following table lists all the keywords in upper case letters.

<table>
<thead>
<tr>
<th>Keywords</th>
<th>AS</th>
<th>EXECUTE</th>
<th>PROCEDURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>FALSE</td>
<td>PUBLIC</td>
<td></td>
</tr>
</tbody>
</table>
The following sections cover the syntactic details of a procedure.

- “Basic Structure of a Procedure” on page 164
- “Compound Statement” on page 167
- “Exceptions” on page 174
- “Transactions” on page 168

### Table 53. SQL Script Keywords

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>FETCH</td>
</tr>
<tr>
<td>CASE</td>
<td>FOR</td>
</tr>
<tr>
<td>CAST</td>
<td>IF</td>
</tr>
<tr>
<td>CLOSE</td>
<td>IMMEDIATE</td>
</tr>
<tr>
<td>COMMIT</td>
<td>IN</td>
</tr>
<tr>
<td>CURRENT_EXCEPTION</td>
<td>INDEPENDENT</td>
</tr>
<tr>
<td>CURSOR</td>
<td>INOUT</td>
</tr>
<tr>
<td>DO</td>
<td>INSERT INTO</td>
</tr>
<tr>
<td>DECLARE</td>
<td>INTO</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>ITERATE</td>
</tr>
<tr>
<td>DELETE</td>
<td>LEAVE</td>
</tr>
<tr>
<td>ELSE</td>
<td>LOOP</td>
</tr>
<tr>
<td>ELSE IF</td>
<td>OPEN</td>
</tr>
<tr>
<td>END</td>
<td>OUT</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>PIPE</td>
</tr>
<tr>
<td></td>
<td>WHEN</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>TYPE</td>
</tr>
<tr>
<td></td>
<td>UNTIL</td>
</tr>
<tr>
<td></td>
<td>UPDATE</td>
</tr>
<tr>
<td></td>
<td>VALUE</td>
</tr>
<tr>
<td></td>
<td>WHEN</td>
</tr>
<tr>
<td></td>
<td>WHILE</td>
</tr>
</tbody>
</table>
Basic Structure of a Procedure

The basic structure of a procedure begins with the word `PROCEDURE` (upper case used here for distinguishing the word), followed by the name of the procedure, an open parenthesis, and a closed parenthesis. The next is a block that begins with the word `BEGIN` and ends with the word `END`. The code for the procedure is placed in the `BEGIN/END` block, as in the following example:

```
PROCEDURE myProcedure ()
BEGIN
    -- Add your code here
END
```

Commenting Code

Notice that the example in the preceding section contains a line that begins with two dashes `--` which indicate that it is a comment line. The comment line is not executed.

There is another way of commenting, which is similar to the style followed in Java programming, as in the following example:

```
PROCEDURE myProc2 ()
BEGIN
    /*
    * This is a multi-line comment
    */
    DECLARE x INTEGER; -- This is a comment
    CALL /shared/procedures/aProcedure(x /* param1*/);
END
```

Statement Delimiter

The statement delimiter is a semicolon `;`. 
Procedure Header

A procedure declaration defines the input parameters and output parameters of the procedure. See “CALL” on page 177 to know how to call a procedure.

**Syntax**

```
PROCEDURE <procedureName> { [ <parameterList> ] } <statement>
```

The parentheses in the procedure’s syntax are optional. If there are parentheses, they can be empty or can contain a list of parameters. A parameter list, indicated by `<paramList>` in the syntax, is a comma-separated list of parameters.

**Parameter Definition**

The syntax for a parameter is as follows.

**Syntax**

```
{ IN | INOUT | OUT } <parameterName> <dataType>
```

**Remarks**

- The data type of a parameter, indicated by `<dataType>` in the syntax, can be any type except ROW listed in “SQL Script Data Types” on page 150.
- You can use any PUBLIC data type defined in the main compound statement within the procedure declaration (indicated by `<compoundStatement>` in the syntax for a procedure). This way a parameter can be defined to be of a named type instead of always being primitive.
Examples

PROCEDURE init_table (IN employee_id INTEGER)
BEGIN
  INSERT INTO T (empid) VALUES (employee_id);
END

PROCEDURE cur_month (OUT x INTEGER)
BEGIN
  SET x = MONTH(CURRENT_DATE());
END

PROCEDURE inc(INOUT x INTEGER)
BEGIN
  SET x = x + 1;
END

PROCEDURE inc(IN x INTEGER)
BEGIN
  SET x = 5; -- Error
END

PIPE - Type Modifier

A type-modifier, named PIPE, is used for streaming a cursor and is used only in procedure parameter declarations to pipeline the output. The PIPE modifier can be applied to any IN or OUT cursor data type, as in the following example:

```
OUT <parameterName> PIPE <cursorDataType>
```

- The PIPE modifier cannot be used on INOUT parameters or on any non-cursor data type.
- An IN parameter with the PIPE modifier can be passed any PIPE variable, which in practice can only come from an IN or OUT parameter of the current procedure. An OUT parameter with the PIPE modifier must be passed a cursor variable with the same schema as the PIPE.
Within a \texttt{PROCEDURE}, a \texttt{PIPE} variable (either \texttt{IN} or \texttt{OUT}) may be used in \texttt{INSERT} statements. See “\texttt{INSERT}” on page 209 for details on \texttt{INSERT}.

Any procedure with the \texttt{PIPE} modifier on an \texttt{OUT} parameter will run on a separate thread, whereas procedures with a \texttt{PIPE} modifier on an \texttt{IN} parameter will not. The calling procedure continues execution as soon as the pipelined procedure begins execution. The calling procedure will find the \texttt{OUT} cursor already initialized, and open the cursor and may fetch from the cursor. See “\texttt{FETCH}” on page 204 for details on \texttt{FETCH}. If the calling procedure accesses any \texttt{NON-PIPE OUT} parameter, however, the calling procedure blocks until the pipelined procedure ends execution. This is because the final value of non-\texttt{PIPE} outputs is not known until the procedure completes.

A \texttt{PIPE} modifier can be in an \texttt{INSERT} statement within an \texttt{EXECUTE IMMEDIATE} statement.

\textbf{Example of PIPE}

```
--Returns a cursor of names all reversed
PROCEDURE reverse_all(OUT result PIPE (rev_name VARCHAR))
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  DECLARE name VARCHAR;
  OPEN c;
  REPEAT
    FETCH c INTO name;
    CALL /shared/reverse(name, name);
    INSERT INTO result (rev_name) VALUES (name);
  UNTIL NOT c.FOUND
END REPEAT;
END
```

\textbf{Compound Statement}

A compound statement must end with a semi-colon if it is not the root statement.
Syntax

```
[label]:
BEGIN
[transactionSpecification]
[declaration]; ...
[statement]; ...
[exceptionBlock]
END [label]
```

Remarks

- The label, indicated by `<label>` in the syntax, is for use with the `LEAVE` statement defined on page 211.
- The label is an optional identifier to name the block. The root `BEGIN` statement (the one directly following the `PROCEDURE` declaration) can have a label.
- If a beginning label is present, the end label is not required. If no beginning label is present, then it is illegal to have an end label. If both the beginning and end labels are present, then both must have the same identifier.
- A compound statement is not required to have any content. It can be completely empty.

Example

```
PROCEDURE init_table()
BEGIN
    DELETE FROM T;
    INSERT INTO T DEFAULT VALUES;
END
```

Transactions

You can declare an independent transaction. See next.
Declaring an Independent Transaction

\texttt{INDEPENDENT [\langle\text{option}\rangle\,...\,]}\,\texttt{TRANSACTION}

Options, indicated by \texttt{[\langle\text{option}\rangle\,...\,]} in the syntax are not case-sensitive.

The following table describes the option flags for an independent transaction:

<table>
<thead>
<tr>
<th>Option Flag</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ROLLBACK_ON_FAILURE</td>
<td>BEST_EFFORT}</td>
</tr>
<tr>
<td>\texttt{COMPENSATE</td>
<td>NOCOMPENSATE}</td>
</tr>
</tbody>
</table>
This flag indicates what the system should do if the server goes down (is interrupted) during the transaction commit when the transaction is partially complete.

The default setting is **IGNORE_INTERRUPT**.

You cannot set two of these flags at the same time.

**IGNORE_INTERRUPT** causes the server to take no special actions. If the server goes down part way through committing a transaction, then no special action will be taken on restart.

**LOG_INTERRUPT** causes the server to store basic transaction information before beginning to commit so it can detect on startup any in-progress transactions and put in log entries about their failure. This option requires two meta-commits per transaction (**start** and **stop**).

**FAIL_INTERRUPT** causes the server to store enough information to perform the requested failure model upon server startup for any in-progress transactions. This option requests meta-commits for start-of-transaction, end-of-transaction, and between each source it commits to. This is expensive.

### Table 54. Option Flags for an Independent Transaction

<table>
<thead>
<tr>
<th>Option Flag</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE_INTERRUPT</td>
<td>This flag indicates what the system should do if the server goes down (is interrupted) during the transaction commit when the transaction is partially complete. The default setting is <strong>IGNORE_INTERRUPT</strong>. You cannot set two of these flags at the same time. <strong>IGNORE_INTERRUPT</strong> causes the server to take no special actions. If the server goes down part way through committing a transaction, then no special action will be taken on restart.</td>
</tr>
<tr>
<td>LOG_INTERRUPT</td>
<td><strong>LOG_INTERRUPT</strong> causes the server to store basic transaction information before beginning to commit so it can detect on startup any in-progress transactions and put in log entries about their failure. This option requires two meta-commits per transaction (<strong>start</strong> and <strong>stop</strong>).</td>
</tr>
<tr>
<td>FAIL_INTERRUPT</td>
<td><strong>FAIL_INTERRUPT</strong> causes the server to store enough information to perform the requested failure model upon server startup for any in-progress transactions. This option requests meta-commits for start-of-transaction, end-of-transaction, and between each source it commits to. This is expensive.</td>
</tr>
</tbody>
</table>
For example, you can use the \texttt{BEST\_EFFORT} and \texttt{NOCOMPENSATE} options as follows in a SQL Script:

```sql
PROCEDURE myProcedure ()
BEGIN INDEPENDENT BEST\_EFFORT NOCOMPENSATE TRANSACTION
  --Add your code here
END
```

\textbf{Remarks}

- The \texttt{BEGIN} statement can be followed by a transaction specifier. See “\texttt{Compound Statement}” on page 167 for information on using \texttt{BEGIN} in a compound statement. If there is no specifier, the block runs within its parent’s transaction and any work it performs is part of the parent transaction.

- When a compound statement is declared as having an independent transaction, all actions in that scope are part of the transaction. See “\texttt{Compound Statement}” on page 167 for information on declaring a compound statement.

- Calling \texttt{COMMIT} is recommended, but is not required. See “\texttt{COMMIT}” on page 182 for using \texttt{COMMIT}.

- Exiting the scope normally will commit the transaction. Exiting the scope through an unhandled exception will cause a roll back on the transaction, but exiting through any handled exception will not implicitly roll back. You must explicitly roll back if that is desired. See “\texttt{ROLLBACK}” on page 218 for details on roll back.

\textbf{Error}

The following table describes the error that may occur while resolving a transaction.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Error Message} & \textbf{Cause} \\
\hline
Conflicting options & Two mutually exclusive options have been declared. \\
\hline
\end{tabular}
\caption{Error Messages for a Transaction}
\end{table}
Compensating Transactions

- The presence of a handler for the `COMPENSATE` exception will cause special behavior at runtime. Unlike other exceptions, this exception will not be handled by an `ELSE` clause. It can only be handled explicitly.
- The `COMPENSATE` exception is special because it is the only exception that can be raised after the compound statement ends. In fact, it may be called a long time after the statement ends. This exception is raised if the transaction is rolled back either explicitly by the transaction’s controller or by the system if a failure occurs during commit.
- The `COMPENSATE` handler has access to all the variables that the block can see, just like any other exception handler. This is a copy of those variables at the time the block exited.
- Compensation can be expensive because this additional storage of variable state has to be kept for every execution of the block. For example, if the block occurs in a loop that ran 1,000 times, there will be 1,000 separate compensation states to run. For this reason, use of the `COMPENSATE` handler has to be watched carefully.
- Only the current local data state is preserved for the handler. The global system state is not preserved. That is, if you call another procedure, it may not be in the same state as it was the first time this block was run. For this reason, any required state should be captured during the normal run into variables so they can be used during the `COMPENSATE` handler.
Examples

PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
<statement>
END

PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
END
--The insert is automatically committed.

PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
  DECLARE my_exc EXCEPTION;
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  RAISE my_exc;
END
--The insert is automatically rolled back.

PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
  DECLARE my_exc EXCEPTION;
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  RAISE my_exc;
EXCEPTION
  ELSE
END
--The insert is automatically committed.
Exceptions

You can define exceptions by providing a unique name to the exception. See also “External Exceptions” on page 175, “Attributes of CURRENT_EXCEPTION” on page 161, and “Raising and Handling Exceptions” on page 174.

Syntax

DECLARE [PUBLIC] <exceptionName> EXCEPTION

An exception may be declared in a sub-scope that has the same name as the one declared in the parent scope. In that case, the one in the parent scope is not visible within the sub-scope.

Raising and Handling Exceptions

Every BEGIN/END block has an optional exception section.

Syntax

BEGIN
    ... ... ... 
    EXCEPTION
        [WHEN <exceptionName>
        [OR <exceptionName> ...]
        THEN <statements> ...]
        [ELSE <statements>]
    END

If the EXCEPTION block is declared, it must have at least one WHEN or an ELSE in it. It does not have to have both WHEN and ELSE, so it is legal to have only an ELSE or only a WHEN. There can be any number of WHEN clauses but only one ELSE clause.

Remarks

- When an exception is raised in a BEGIN/END block, the first exception-handler WHEN clause that matches the exception is executed.
- All variables from the scope are available within the exception handler. This technique is different from Java, for example. In Java, nothing from the try block is available in the catch block. In SQL Script, all variables available
within the `BEGIN` area are available within the `EXCEPTION` area. They don’t go out of scope until the `END` is reached.

- If an exception is not handled within a block, that block leaves scope as with a `LEAVE` statement and the same exception is raised in the parent scope, where it may be handled. If there are no further scopes, the exception is thrown out of the procedure to the caller. If the caller is SQL Script, they receive this error. If the caller is JDBC or a Java Procedure, then a Java exception is received. If the caller is in a SQL `FROM` clause, then the statements ends with a runtime exception.

- Any exception raised while in an exception handler, immediately leaves the current scope as if it were an unhandled exception in this scope.

- Use the `RAISE` statement to re-raise an exception.

**Example**

```sql
PROCEDURE p (IN x INTEGER, OUT result BIT)
BEGIN
  DECLARE illegal_arg_ex EXCEPTION;
  ...
  IF x < 0 THEN
    RAISE illegal_arg_ex;
  END
  SET result = 1;  --success
EXCEPTION
  WHEN illegal_arg_ex THEN
    SET result = 0;  --failure
END
```

**External Exceptions**

System exceptions are considered to be globally reserved names. An exception can also be used from another procedure if the exception is public.

**Syntax**

```sql
<compNamespacePath>.<exceptionName>
```
A Composite namespace path, as indicated by `<compNamespacePath>` in the syntax, is a sequence of elements separated by a forward slash (`/`).

**Statement Reference**

The following table lists all the SQL Script statements:

<table>
<thead>
<tr>
<th>Statement and Reference</th>
<th>Statement and Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BEGIN...END (Compound Statement)&quot;</td>
<td>&quot;IF&quot; on page 208</td>
</tr>
<tr>
<td>on page 177</td>
<td>&quot;INSERT&quot; on page 209</td>
</tr>
<tr>
<td>&quot;CALL&quot; on page 177</td>
<td>&quot;ITERATE&quot; on page 210</td>
</tr>
<tr>
<td>&quot;CASE&quot; on page 180</td>
<td>&quot;LEAVE&quot; on page 211</td>
</tr>
<tr>
<td>&quot;CLOSE&quot; on page 182</td>
<td>&quot;LOOP&quot; on page 212</td>
</tr>
<tr>
<td>&quot;COMM&quot; on page 182</td>
<td>&quot;OPEN&quot; on page 213</td>
</tr>
<tr>
<td>&quot;DECLARE CONSTANT&quot; on page 183</td>
<td>&quot;PATH&quot; on page 214</td>
</tr>
<tr>
<td>&quot;DECLARE CURSOR&quot; on page 184</td>
<td>&quot;RAISE&quot; on page 215</td>
</tr>
<tr>
<td>&quot;DECLARE EXCEPTION - public&quot; on page 189</td>
<td>&quot;REPEAT&quot; on page 217</td>
</tr>
<tr>
<td>&quot;DECLARE TYPE&quot; on page 191</td>
<td>&quot;ROLLBACK&quot; on page 218</td>
</tr>
<tr>
<td>&quot;DECLARE Variable&quot; on page 192</td>
<td>&quot;SELECT INTO&quot; on page 218</td>
</tr>
<tr>
<td>&quot;DECLARE VECTOR&quot; on page 194</td>
<td>&quot;SELECT INTO&quot; on page 218</td>
</tr>
<tr>
<td>&quot;DELETE&quot; on page 202</td>
<td>&quot;SET&quot; on page 219</td>
</tr>
<tr>
<td>&quot;EXECUTE IMMEDIATE&quot; on page 203</td>
<td>&quot;UPDATE&quot; on page 220</td>
</tr>
<tr>
<td>&quot;FETCH&quot; on page 204</td>
<td>&quot;WHILE&quot; on page 221</td>
</tr>
<tr>
<td>&quot;FOR&quot; on page 205</td>
<td></td>
</tr>
</tbody>
</table>
BEGIN...END (Compound Statement)

The syntax for a compound statement is as follows:

```
[label]:
BEGIN
[<transactionSpecification>]
[<declaration>; ...]
[<statement>; ...]
[<exceptionBlock>]
END [label]
```

Remarks

- The order of the parameters in the procedure’s declaration is important. While it is conventional to list IN, then INOUT, then OUT parameters in that order, they can be commingled.
- IN parameters are unchangeable in the procedure (like a const parameter).
- OUT parameters are initialized to NULL within the procedure. Setting a value into an OUT parameter assigns the value to the variable in the caller.
- INOUT parameters are like OUT parameters that are pre-initialized by the caller. Any calling environment that does not have variables should treat these parameters as if they were a pair of IN and OUT parameters.

CALL

The CALL statement is used to call a procedure.

Syntax

```
CALL <procedureName> [ [valueExpression][, ...] ]
```

- `<procedureName>` in the syntax refers to the name of the procedure declared using the syntax for a procedure declaration. See “Procedure Header” on page 165 for procedure declaration.
- Parentheses in the CALL syntax are not required if there are no parameters. See procedure declaration for details on parameter declaration.
Remarks

- IN parameters can be passed any value expression. For details on value expressions, see “Value Expressions” on page 154. The expression will be implicitly cast if required to match the type of the IN parameter. IN parameters can be literals, expressions, or variables. If an IN parameter is a variable, the value will not be altered. IN parameters with the PIPE modifier (described on page 166) can only pass in variables that are also PIPE variables. In practice this means only IN or OUT parameters of the current procedure that have the PIPE modifier can be passed in.

- The expressions being passed to IN parameters will be evaluated from left to right.

- INOUT and OUT parameters must be passed a variable of the appropriate type. No implicit type conversion will be supported. For INOUT parameters, the value is not altered if it is not changed in the procedure. For OUT parameters, the value is set to NULL if not altered in the procedure. OUT parameters with the PIPE modifier can only be passed a cursor variable with the same cursor type as the PIPE.
**Examples**

PROCEDURE square (IN x INTEGER, OUT result INTEGER)
BEGIN
    SET result = x * x;
END

PROCEDURE p()
BEGIN
    DECLARE y INTEGER;
    CALL square(2, y);
    -- y is 4
    CALL square(y, y);
    -- y is 16
END

PROCEDURE factorial (IN x INTEGER, OUT result INTEGER)
BEGIN
    IF x = 1 THEN
        SET result = 1;
    ELSE
        CALL /shared/factorial(x-1; result);
        SET result = x * result;
    END
END
CASE

There are two legal formats for CASE statements.

**Syntax 1**

CASE <valueExpression>
    WHEN <valueExpression> THEN <statements>
    [...] [ELSE <statements>] END AS <new_column_name>

The above format is used to evaluate an expression once, then find a matching value. The WHEN clauses are evaluated in order and the first match is used.

**Syntax 2**

CASE
    WHEN <conditionalExpression> THEN <statements>
    [...] [ELSE <statements>] END AS <new_column_name>

The above format is used to evaluate a series of tests like an IF/THEN/ELSEIF/ELSE. The WHEN clauses are evaluated in order and the first match is used.

**Remark**

- There may be zero or more statements in the statements area indicated by <statements>.
Examples

PROCEDURE get_month_name(OUT month_name VARCHAR)
BEGIN
CASE MONTH(CURRENT_DATE())
    WHEN 1 THEN
        SET month_name = 'JAN';
    WHEN 2 THEN
        SET month_name = 'FEB';
    WHEN 3 THEN
        SET month_name = 'MAR';
    ...;
    WHEN 11 THEN
        SET month_name = 'NOV';
    WHEN 12 THEN
        SET month_name = 'DEC';
END CASE;
END

PROCEDURE get_duration(IN seconds INTEGER, OUT result VARCHAR)
BEGIN
CASE
    WHEN seconds < 60 THEN
        SET result = CAST (
            CONCAT(seconds, ' seconds') AS VARCHAR);
    WHEN seconds < 60*60 THEN
        SET result = CAST (
            CONCAT(seconds/60, ' minutes') AS VARCHAR);
    ELSE
        SET result = CAST (
            CONCAT(seconds/3600, ' days') AS VARCHAR);
END CASE;
END
CLOSE

The `CLOSE` statement is used to close a cursor. See “DECLARE CURSOR” on page 184 for details on declaring cursors.

**Syntax**

```sql
CLOSE <cursor>
```

**Errors**

The following table describes the errors that may occur while executing a `CLOSE` statement.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized cursor</td>
<td>A cursor variable is used and is not initialized at the time it is opened.</td>
</tr>
<tr>
<td>Cursor is not open</td>
<td>The cursor is closed when it is not open.</td>
</tr>
</tbody>
</table>

COMMIT

The `COMMIT` statement is used to commit an independent transaction inside a compound statement. See “Transactions” on page 168 for details on transactions, and “Compound Statement” on page 167 for details on compound statements.

**Syntax**

```sql
COMMIT
```

**Remark**

- It is illegal to call `COMMIT` in a compound statement that is not declared independent.
Example

PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
    DECLARE my_exec EXCEPTION;
    INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
    COMMIT;
    RAISE my_exec;
END

DECLARE CONSTANT

You can define constants by providing a unique name to each constant.

Remarks

- PUBLIC CONSTANT should be declared at a global level.
- Wherever you can use a literal, you can use a CONSTANT. It is not modifiable.
- Variable declaration rules apply in the case of CONSTANT.
- Declare the CONSTANT first before using it.

Syntax

DECLARE [PUBLIC] <constantName> TYPE DEFAULT

Example

PROCEDURE constants ( )
BEGIN
    DECLARE PUBLIC x CONSTANT INT DEFAULT 1234;
    DECLARE PUBLIC y CONSTANT ROW (a INT, b CHAR) DEFAULT (1, 'abc');
    END
DECLARE CURSOR

You can define your own variable/static type cursors.

Variable Cursor

You can define a new cursor variable by providing a unique name and optionally specifying its data type. See “Attributes of Cursors” on page 159, “OPEN” on page 213, “FETCH” on page 204, and “CLOSE” on page 182 for additional information on cursors.

Syntax

DECLARE <variableName> CURSOR [<dataType>]

The data type, indicated by <dataType> in the syntax, is optional and can be a named ROW data type or the syntax for a ROW data type.

Remarks

- When declared, the cursor variable is initialized to NULL. It cannot be initialized to any other value at declaration.
- You can use the SCROLL keyword in an OPEN statement to open a cursor after a row has been fetched from a cursor, as follows:

  DECLARE i INT;
  DECLARE x CURSOR (a int) FOR SELECT COUNT(*) FROM /services/databases/system/ALL_USERS;
  OPEN x SCROLL;
Examples

--Returns the first name
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
    DECLARE c CURSOR (name VARCHAR);
    OPEN c FOR SELECT name FROM /shared/T;
    FETCH c INTO p_name;
    CLOSE c;
END

PROCEDURE p (OUT p_name VARCHAR)
BEGIN
    DECLARE c CURSOR (name VARCHAR);
    OPEN c FOR SELECT name FROM /shared/T;
    CLOSE c;

    --Reopen with the same query
    OPEN c;
    CLOSE c;

    --Reopen with new query
    OPEN c FOR SELECT name FROM /share/U WHERE birthdate > 2000-01-01';
    CLOSE c;
END

Static Cursor

You can define a static cursor by providing a unique name for it and specifying the query expression associated with the cursor.
Syntax

DECLARE <cursorName> CURSOR FOR <queryExpression>

Name resolution works like a standalone SELECT statement. Variables may not be used in the query expression. Bind variables (such as '?') may not be used.

Remarks

- Declaring a static cursor is logically equivalent to preparing a statement in JDBC.
- A cursor declared this way cannot have its value changed. It is like a constant in this regard.
Examples

PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
END

--Returns the first name
PROCEDURE p (OUT p_name VARCHAR)
BEGIN
  DECLARE c CURSOR FOR SELECT name FROM /shared/T;
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
...

  --Reopen cursor
  OPEN c;
  FETCH c INTO p_name;
  CLOSE c;
END

PROCEDURE p
BEGIN
  DECLARE c CURSOR (name VARCHAR);
  DECLARE d CURSOR FOR SELECT name FROM /shared/T;

  --Open a new cursor in cursor variable c
  OPEN c FOR SELECT name FROM /shared/T;

Assign the cursor referred to by d to c
The original cursor referred to by c will no longer be accessible

```sql
SET c = d;
```

--c and d cursor variables now refer to the same cursor
--Use either one to open the cursor

```sql
OPEN d; -- or OPEN c
--c.ISOPEN is true
```

--Returns an opened cursor
```sql
PROCEDURE p (OUT p_cursor CURSOR (name VARCHAR))
BEGIN
    DECLARE c CURSOR FOR SELECT name FROM /shared/T;
    SET p_cursor = c;
    OPEN p_cursor;
END
```

--Returns an opened static cursor
```sql
PROCEDURE p (OUT p_cursor CURSOR (name VARCHAR))
BEGIN
    OPEN p_cursor FOR SELECT name FROM /shared/T;
END
```

```sql
PROCEDURE p (OUT p_id INTEGER, OUT p_name VARCHAR)
BEGIN
    DECLARE c CURSOR FOR SELECT id, name FROM /shared/T;
    DECLARE r ROW (id INTEGER, name VARCHAR);
```
OPEN c;
FETCH INTO c;
CLOSE c;

SET p_id = r.id;
SET p_name = r.name;
END

PROCEDURE p ( )
BEGIN

DECLARE TYPE r_type ROW (id INTEGER, name VARCHAR);
DECLARE c CURSOR r_type;
DECLARE r r_type;

OPEN c FOR SELECT id, name FROM /shared/T;
FETCH INTO c;
CLOSE c;

DECLARE EXCEPTION - public

You can define exceptions by providing a unique name to each exception. See also “External Exceptions” on page 175, “Attributes of CURRENT_EXCEPTION” on page 161, and “Raising and Handling Exceptions” on page 174.

Syntax

DECLARE [PUBLIC] <exceptName>
EXCEPTION

An exception may be declared in a sub-scope that has the same name as the one declared in the parent scope. In that case, the one in the parent scope is not visible within the sub-scope.
Remark

- The `PUBLIC` keyword can only be used in the root compound statement of a `PROCEDURE`. It makes the exception visible outside the procedure as described in the section “External Exceptions” on page 175. See “Compound Statement” on page 167 for information on compound statements.

Example

```sql
PROCEDURE f(IN x INTEGER)
BEGIN
    DECLARE PUBLIC illegal_arg_ex EXCEPTION;

    IF x IS NULL THEN
        RAISE illegal_arg_ex;
    END IF;
...
END

PROCEDURE p(IN x INTEGER, IN result BIT)
BEGIN
    CALL /shared/f(x);
    SET result = 1; -- success
EXCEPTION
    WHEN /shared/f.illegal_arg_ex THEN
        SET result = 0; -- failure
END
```
DECLARE TYPE

The data types supported in SQL Script are listed in the section “Data Types” on page 149.

You can also declare a new data type.

**Syntax**

```
DECLARE [PUBLIC] TYPE <typeName> <dataType>
```

where <dataType> can be a ROW type or regular data type.

Defining a new data type is effectively a way to create an alias for a data type.

The declaration can be used to make a custom string, such as aliasing FirstName to VARCHAR(24), or (more likely) for making an alias for a column set, such as aliasing ResponseCursorType to ROW(col1 VARCHAR(40), col2 INTEGER).

**Remarks**

- You can use DECLARE TYPE on CURSOR types, as in

  ```
  DECLARE PUBLIC TYPE cursor_datatype_exampleA
  CURSOR (fieldA INTEGER, fieldB VARCHAR(255), fieldC DATE)
  ```

- If you alias ID to be of type INTEGER then it is a distinct type and is not a plain integer any more.

- To make the data types visible outside of a procedure, the PUBLIC keyword can only be used in the root compound statement of a procedure.
Examples

PROCEDURE p ( )
BEGIN
  DECLARE TYPE name_type VARCHAR(50);
  DECLARE TYPE money_type DECIMAL(18, 2);
  DECLARE TYPE id_type BIGINT;

  DECLARE a name_type DEFAULT 'Joe';
  DECLARE b money_type DEFAULT 12.34;
  DECLARE c id_type DEFAULT 1234567890;
  ...
END

PROCEDURE p ( )
BEGIN
  DECLARE TYPE r_type ROW (i INTEGER, name VARCHAR, birthdate DATE);
  DECLARE r r_type;
  DECLARE s r_type;

  SET r.id = 123;
  SET r.name = '5';
  SET r.birthdate = '1990-10-31';
  ...
END

DECLARE Variable

You can define a non-cursor variable by specifying its name and data type, and initializing it with a default value. See "DECLARE CURSOR" on page 184 for defining cursor variables.

Syntax

DECLARE <variableName>[,...] <dataType> DEFAULT <valueExpression>
The **DEFAULT** syntax is used to initialize a variable.

**Remarks**
- You can declare more than one variable at a time, provided all the variables are of the same data type but each one has a unique name.
- The value expression, indicated by `<valueExpression>` in the syntax, can use `IN` parameters, previously declared variables in this block, and any variables in parent blocks. In the current block, the value expression cannot use variables that are defined later. If the value expression’s type does not match the variable’s type, an implicit cast will be performed (if possible). See “Procedure Header” on page 165 for information on `IN` parameters.
- Any variable that is not initialized with a **DEFAULT** clause has the value **NULL**.
- If the evaluation of the value expression causes an exception, any other declared variables that have not yet been initialized are set to **NULL** before entering the exception handler.
DECLARE VECTOR

VECTOR is a collection data type. It is expandable, ordered, typed, and requires a data type at initialization.

This section provides the general syntax for a VECTOR and describes the functionality of VECTORS in SQL Script. Examples are given at the end of the section.

Syntax

The syntax for a VECTOR collection is as follows:

```
DECLARE <identifier> VECTOR (<data type>) DEFAULT VECTOR [<value>, <value>]
```

where the DEFAULT clause is optional and may be used to initialize the VECTOR with values.

Remarks

This section gives the details on VECTORS.

Base Data Types

- A VECTOR cannot be the base data type of another VECTOR, so you cannot use the following declaration:

  ```
  DECLARE myvector VECTOR (VECTOR (CHAR));
  ```

- ROW is an acceptable base data type of a VECTOR, and is necessary for any implementation of collections, as in the following example:

  ```
  DECLARE myVector VECTOR(ROW (a INTEGER, b INTEGER, c CHAR, d CHAR));
  ```

- ROWs may also contain VECTORS, and a field in the ROW can be accessed through the dot notation as follows:

  ```
  DECLARE myRow ROW(a INTEGER, v VECTOR(INTEGER));
  SET myRow = ROW(1, VECTOR[9,10,11]);
  set myRow.v[2] = 9;
  ```

  ```
  DECLARE vecRow VECTOR(ROW (a INTEGER, b CHAR));
  SET vecRow = VECTOR[{22, 'text'}];
  ```
SET vecRow[1].a = vecRow[1].a + 15;

**Declaration**

- **You cannot** declare a VECTOR as a field in a CURSOR or a PIPE, so the following declaration would **not** be permitted:

  DECLARE myCursor CURSOR (a VECTOR(CHAR));

- VECTORS can be declared as PUBLIC CONSTANTS or non-public CONSTANTS. The contents of such VECTORS should not be modified.

- The initial contents of a CONSTANT VECTOR must be defined in a `DEFAULT` clause and must be literals or references to other similar type of VECTORS.

**Assigning Values to VECTOR Elements**

- An empty VECTOR with no base type can be created by the expression `VECTOR []`.

- Elements in a VECTOR may be assigned a value of `NULL`.

  ```
  SET myvector[1] = NULL;
  ```

- The VECTOR is set to `NULL` at declaration and must be initialized before it can be used, as in the following example. Any reference to an un-initialized VECTOR will result in an error.

  ```
  VECTOR ['my text', 'your text']
  ```

  This expression may be assigned to a compatible VECTOR with the `SET` statement, as follows:

  ```
  SET myvector = VECTOR ['my text', 'your text'];
  SET yourvector = VECTOR [ROW(2,3), ROW(4,5)];
  SET yourvector = myvector;
  ```

  In the above declaration, the contents of the source vector `yourvector` will be copied to the target vector `myvector`, and the target vector will be initialized.

- Vectors may be used as parameters in procedures, and the procedures with `OUT` or `INOUT` parameters may alter the vector in the same manner as the `SET` statement.

  ```
  CALL myProcedure(myvector);
  ```

- Once spaces are allocated in a VECTOR by initializing the VECTOR, elements in the VECTOR can be accessed through square brackets, as in arrays in other programming languages. VECTOR indexes start at 1 and increment by 1.
SET myvector[20] = 'my text';
SET yourvector[2 + index] = myvector[20];

- A VECTOR index must evaluate to a numeric value. Otherwise, there will be an error, as in the following example:
  SET yourvector[1 || 'text'] = 'text';
- If a VECTOR index evaluates to NULL, the element reference will result in NULL.
- If the target reference index is NULL, that will result in an error, as in the following example:
  SET myvector(NULL) = 'text';

VECTORS are bound by the current allocation, but can be resized through reassignment or through system procedures.

VECTORS may be assigned to other VECTORS that have implicitly assignable data types. In the case where the data type is not the same, a VECTOR will be created, and all elements will automatically have the CAST function run to convert the value to the target type.

**Comparing VECTORS**

- VECTORS may be compared to one another if their base types are comparable. Only comparison operators such as = (equal to) and != (not equal) are supported now.
- VECTORS are equal if they have the same number of values, and corresponding elements are equal. If either VECTOR is NULL, the result of the comparison is unknown. If any of the elements is NULL, the result of the comparison is unknown.

**VECTORS and Functions**

- Several functions are available to modify the contents of a VECTOR. Currently, the following functions are supported: CARDINALITY, CONCAT, ||, CAST, EXTEND, and TRUNCATE. All VECTORS, regardless of their base data type are accepted as arguments for these functions, which are described next.
**CARDINALITY** — This function returns the number of elements allocated in the VECTOR.

**CAST** — This function converts all the elements in a VECTOR to the desired target data type. The result VECTOR is of the same size as that of the source VECTOR. If the VECTOR has a NULL element, the result VECTOR will contain NULL. The source VECTOR’s data type and the target VECTOR’s data type must be compatible. For details on data types that are compatible for casting, see the section “CAST” on page 40.

**CONCAT** — This function and the `||` operator add two vectors that have the same data type together. If either of the VECTORs is NULL, an error occurs indicating that the resultant VECTOR is NULL. Concatenating non-NULL VECTORs result in a new VECTOR containing the elements from the concatenated VECTORs. The elements of the input VECTORs are added successively; that is, the elements of the first VECTOR populates the result VECTOR first, the elements of the second VECTOR populates the result VECTOR subsequently, and so on.

**EXTEND** — This function appends the specified number of elements to a VECTOR. The appended number of elements will be assigned a NULL value, and the syntax is as follows:

```
SET myvector = EXTEND (myvector, 2);
```

- If the number of elements specified to be appended evaluates to NULL, this function will return NULL.
- If the VECTOR is NULL, an error will occur indicating that the VECTOR is NULL.
- If the specified number is a negative number, an error would occur.

**FIND_INDEX** — The function searches a VECTOR for the first occurrence of a specified value. It accepts two arguments. The first argument is any scalar value. The second argument is the VECTOR that is searched. The index starts at 1.

- The base type of the VECTOR and the supplied argument’s data type must be comparable or implicitly castable.
- If the searched value is not found in the VECTOR, the result will be zero.
- If either the VECTOR or the supplied argument is NULL, the result of the function will be NULL.
Example
DECLARE v VECTOR(INT) DEFAULT VECTOR {11, 22, 33, 44};
SET i = FIND_INDEX(22, v);

will return a value of 2.

TRUNCATE — This function removes the specified number of elements from a
VECTOR. The syntax is as follows:
SET myvector = TRUNCATE (myvector, 2);
- If the number of elements specified to be appended evaluates to NULL, this
  function will return NULL.
- If the specified number of elements to be removed is negative, an error
  would occur.
- If the specified number of elements to be removed exceeds the initial size
  of the VECTOR, an error would occur.
- If the VECTOR is NULL, an error will occur indicating that the VECTOR is NULL.

Examples
This section contains several examples to illustrate the functionality of VECTORS in
SQL Script.
PROCEDURE vectorExample1()
BEGIN
  DECLARE vec1 VECTOR(ROW(a int, b char));
  DECLARE vec2 VECTOR(ROW(x int, y char));
  SET vec1 = VECTOR{(11, 'one in vec1'), (12, 'two in vec1')};
  SET vec2 = VECTOR{(21, 'one in vec2'), (22, 'two in vec2')};
  CALL print(vec1[1].b);
  CALL print(vec1[2].b);
  IF vec1 != vec2 THEN
    CALL print(vec2[1].y);
  END IF;
END

PROCEDURE vectorExample2()
BEGIN
  DECLARE vec1 VECTOR(ROW(a int, b char));
  DECLARE vec2 VECTOR(ROW(x int, y char));

  SET vec1 = VECTOR[(11, 'one in vec1'), (12, 'two in vec1')];
  SET vec1[1].a = vec1[1].a + 11;
  SET vec2 = VECTOR[(5, 'one in vec2'), (10, 'two in vec2')];
  SET vec1 = vec2;
  CALL PRINT(TO_CHAR(vec1[2].a));
END

PROCEDURE vectorExample3(OUT x VECTOR(INTEGER))
BEGIN
  DECLARE myvector VECTOR(INTEGER);

  SET x = VECTOR[5, 55, 60];
  SET myvector = x;
  CALL PRINT(TO_CHAR(x[1]));
END

PROCEDURE vectorExample4()
BEGIN
  DECLARE vConst1 CONSTANT VECTOR(INTEGER)
  DEFAULT VECTOR[1, 2];
  DECLARE vConst2 CONSTANT VECTOR(INTEGER)
  DEFAULT VECTOR[99, vConst1[2]]
  DECLARE x INTEGER;
  DECLARE y INTEGER;
SET x = vConst1[1];
SET y = vConst2[1];
CALL PRINT(TO_CHAR(x));
CALL PRINT(TO_CHAR(y));
END

PROCEDURE vectorExample5()
BEGIN
    DECLARE PUBLIC vConst1 CONSTANT VECTOR(INTEGER)
        DEFAULT VECTOR[1, 2];
    DECLARE PUBLIC vConst2 CONSTANT VECTOR(INTEGER)
        DEFAULT VECTOR[99, vConst1[2]];  
    DECLARE x INTEGER;
    SET x = vConst2[2];
    CALL PRINT(TO_CHAR(x));
END

PROCEDURE vectorExample6(OUT Name VECTOR(CHAR(255)))
BEGIN
    DECLARE firstName VECTOR(CHAR);
    DECLARE lastName VECTOR(CHAR);
    SET firstName = VECTOR['john'];
    SET lastName = VECTOR['doe'];
    SET Name = CONCAT(firstName, lastName);
END

PROCEDURE vectorExample7(OUT card INTEGER)
BEGIN
    DECLARE myvector VECTOR(INTEGER);

200
SET myvector = VECTOR[5, 55, 19, 15, 23];
SET card = CARDINALITY (myvector);
END

PROCEDURE vectorExample8(OUT ext VECTOR(INTEGER))
BEGIN
  DECLARE myvector VECTOR(INTEGER);
  DECLARE NEWVECTOR VECTOR(INTEGER);
  SET myvector = VECTOR[5, 55, 19, 15, 23];
  SET myvector = EXTEND(myvector, 2);
  SET ext = myvector;
END

PROCEDURE vectorExample9(OUT ext VECTOR(INTEGER))
BEGIN
  DECLARE myvector VECTOR(INTEGER);
  SET myvector = VECTOR[5, 55, 19, 15, 23];
  SET myvector = VECTOR[NIL];
  SET myvector = EXTEND(myvector, 2);
  SET ext = myvector;
END

PROCEDURE vectorExample10(OUT trunc VECTOR(INTEGER))
BEGIN
  DECLARE myvector VECTOR(INTEGER);
  DECLARE newvector VECTOR(INTEGER);
  SET myvector = VECTOR[5, 55, 19, 15, 23];
  SET newvector = TRUNCATE(myvector, 2);
SET trunc = newvector;
END

PROCEDURE vectorExample11(OOUT trunc VECTOR(INTEGER))
BEGIN
DECLARE myvector VECTOR(INTEGER);
DECLARE newvector VECTOR(INTEGER);

SET myvector = VECTOR[5, 25, 30];
SET newvector = TRUNCATE(myvector, NULL);
SET trunc = newvector;
END

DELETE

Any legal DELETE statement that the system accepts can be used as a standalone SQL Script statement. For details on acceptable DELETE statements, refer to Appendix A.

**Syntax**

DELETE FROM <table> [WHERE <conditionalExpression>]

**Remark**

- Variables are allowed in a SQL statement anywhere a literal is allowed.
Examples

PROCEDURE p ( )
BEGIN
    DELETE FROM /shared/scores;
    INSERT INTO /shared/scores VALUES ('Joe', 1001);
    UPDATE /shared/.scores SET score=1239 WHERE name='Sue';
END

PROCEDURE p (IN p_name VARCHAR, IN new_score)
BEGIN
    DELETE FROM /shared/scores WHERE name=p_name;
    INSERT INTO /shared/scores VALUES (p_name, new_score);
    UPDATE /shared/.scores SET score=new_score WHERE name=p_name;
END

PROCEDURE p (IN y VARCHAR)
BEGIN
    --T has columns x and y
    --The following y refers to the column, not the parameter
    DELETE FROM /shared/T WHERE x = y;
END

EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement is used to dynamically execute certain SQL statements.

Syntax

EXECUTE IMMEDIATE <valueExpr>

The value expression, indicated by <valueExpr> in the syntax, must evaluate to a string type (CHAR or VARCHAR). The text in this string is executed as SQL.
Remarks

- This form of dynamic SQL is useful mainly for INSERT, UPDATE, and DELETE statements. It has no value for SELECT since the selections cannot be assigned to anything. See the OPEN FOR statement used in “OPEN” on page 213 to know how to perform a dynamic SELECT.

Example

```
PROCEDURE drop (IN table_name VARCHAR)
BEGIN
    DECLARE sql_stmt VARCHAR;
    SET sql_stmt = CAST(CONCAT('DELETE FROM '; table_name) AS VARCHAR);
    EXECUTE IMMEDIATE sql_stmt;
END
```

FETCH

The FETCH statement is used to read one row from an open cursor.

Syntax

```
FETCH <cursor> INTO <varList>
```

The variable list, indicated by `<varList>` in the syntax, works like the SELECT INTO clause. See “SELECT INTO” on page 218 for details.

The variable list can be a list of variables (same number as the number of projections) or a ROW variable with the right schema. See “DECLARE CURSOR” on page 184 for information on ROW.

Remarks

- It is illegal to fetch from a cursor that is not open.
- Fetching past the last row does not cause an error. The variables are not altered and the FOUND attribute is set to false. See Table 51, “Attributes of Cursors” on page 160 for details on FOUND.
You can specify the direction of the fetch to be *NEXT* or *FIRST*. These words must be used along with the keyword *FROM*, as follows:

```
FETCH NEXT FROM x INTO i;
FETCH FIRST FROM x INTO i;
```

If no fetch orientation is specified, *NEXT* is the default.

If the orientation is *NEXT*, the fetch will behave as it always has. It will obtain the current row’s data into the target variables.

If *FIRST* is specified as the orientation, the cursor must be a *SCROLL* cursor, otherwise an error will result. See Remarks in “DECLARE CURSOR” on page 184.

If the orientation specified is *FIRST*, the cursor will be repositioned to the first row, and the first row’s data will be placed into the target variables.

**Errors**

The following table describes the errors that may occur while executing a *FETCH* statement.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized cursor</td>
<td>The cursor variable is used but is not initialized at the time it is fetched.</td>
</tr>
<tr>
<td>Cursor is not open</td>
<td>Cursor is closed when the fetch is attempted.</td>
</tr>
</tbody>
</table>

**FOR**

The *FOR* statements are used to loop through a query or cursor. There are two different formats of *FOR* statements.
Syntax1

```
[label]:]
FOR <loopVariable> AS [cursorName> CURSOR FOR
<queryExpression> DO
<statements>
END FOR [label]
```

The above format is used to loop across a query expression.

Syntax2

```
[label]:]
FOR <loopVariable> AS <cursorVariable> DO
<statements>
END FOR [label]
```

The above format is used to loop across a cursor. See “DECLARE CURSOR” on page 184 for information on cursors.

The label, indicated by <label> in the syntax, is an optional identifier to name the block. This is for use with the LEAVE and ITERATE statements. See “LEAVE” on page 211 and “ITERATE” on page 210.

If a beginning label is present, the end label is not required. If no beginning label is present, then it is illegal to have an end label. If both the beginning and end labels are present, then both must have the same identifier.

There may be zero or more statements in the <statements> area.

Remarks

- The FOR statement declares the loop variable to be of the proper type to match the query expression (a ROW). You do not have to declare that variable elsewhere. The variable is only legal within the loop block. This variable may have the same name as another variable in the current scope (or a parent scope), but it cannot have the same name as a parameter to the procedure. If it does have the same name, the same rules apply as for declaring variables in a compound statement. See “Compound Statement” on page 167 for details on compound statements.
If a cursor variable is provided in the first format (Syntax 1), it is also declared at this point. You do not declare it separately. This variable is set to be a cursor for the provided query expression.

The cursor is opened when it starts. You do not have to open the cursor. It then fetches rows (use FETCH) one at a time and assigns the row into the loop variable. This makes it possible to operate on each row one at a time. The cursor is closed automatically when the loop ends. See “FETCH” on page 204. If you open the cursor (and even fetch a few rows), the FOR loop will pick up where the cursor currently is. If you don’t open the cursor, the FOR statement will open it for you.

The FOR loop will close the cursor no matter how the loop exits (even with a LEAVE statement).

It is always an error to CLOSE an already closed cursor. So, you will get an error if you try to close a cursor that was used by a FOR loop.

When a FOR loop is passed a cursor, it will open the cursor if it is not already open, but it is fine if the cursor is already open.

After the FOR loop, the cursor will be closed. Even if you try to LEAVE the FOR loop, the cursor will be closed.

If you try to CLOSE a closed cursor, you will get an error.
Example
--Returns the average of all scores
PROCEDURE avr_score(OUT result INTEGER)
BEGIN
  DECLARE crs CURSOR FOR
    SELECT name, score FROM /shared/U ORDER BY score DESC;
  DECLARE total INTEGER DEFAULT 0;
  DECLARE cnt INTEGER DEFAULT 0;
  OPEN crs;
  FOR r AS crs DO
    SET total = total + r.score;
    SET cnt = cnt + 1;
  END FOR;
  SET result = total/cnt;
END

IF
The IF statement is used to evaluate a condition.

Syntax
IF <conditionalExpression> THEN
  <statements>
[ELSEIF
  <statements>
[ELSE <statements>]
END IF

The <statements> areas are sequences of statements. There may be zero or more statements in a statement sequence. Each statement ends with a semicolon.
Example

PROCEDURE "max" (IN a INTEGER, IN b INTEGER, OUT "max" INTEGER)
BEGIN
  IF a IS NULL OR b IS NULL THEN
    SET "max" = NULL;
  ELSEIF a > b THEN
    SET "max" = b;
  ELSEIF b > a THEN
    SET "max" = b;
  ELSE
    SET "max" = a;
  END IF;
END

INSERT

The INSERT INTO statement is used to insert values into the columns of the table specified. Most any INSERT statement can be used as a standalone SQL Script statement.

Variables are allowed in a SQL statement anywhere a literal is allowed.

Syntax

INSERT INTO table_name[(column_A, column_X,...)] VALUES
  ('value1', 'value X', ...);

Remarks

- Specification of the column names is optional, as is indicated by the square bracket enclosure in the syntax example above. The VALUES list are comma separated values for insertion into the specified columns.
- The INSERT INTO statement may also be used to insert a complete row of values without specifying the column names. Values must be specified for every column in the table in the order specified by the DDL. If the number of values is not exactly the same as the number of columns in the table or if a value is not allowed for a particular data type, an exception will be thrown.
The syntax of `INSERT` is extended to allow `PIPE` variables to be used where a table name is normally used. This is how rows are inserted into a `PIPE`. See “`PIPE - Type Modifier`” on page 166.

**Examples**

```sql
PROCEDURE p1 (OUT result PIPE(C1 VARCHAR(256))
BEGIN
  INSERT INTO result(C1) VALUES(some_variable);
END

PROCEDURE p2 ( )
BEGIN
  INSERT INTO birthdays(person_name,"birth date","annotation")
  VALUES('Chris Smith','2006-12-20','Last years gift:Watch');
END
```

**ITERATE**

The `ITERATE` statement is used to continue the execution of the specified label.

**Syntax**

```
ITERATE <label>
```

**Remark**

- The `ITERATE` statement is equivalent to `continue` in Java. It jumps to the end of the loop block and causes the loop to evaluate its condition (if available) and/or loop back to the top.
**Example**

```sql
PROCEDURE
BEGIN
    DECLARE c CHAR(1);
    DECLARE ix INTEGER DEFAULT 1;
    SET result = ' ';

    label a:
    WHILE ix <= LENGTH(s) DO
        SET c = CAST(SUBSTRING(s, ix, 1) AS CHAR(1));
        SET ix = ix + 1;
        IF c = ' ' THEN
            ITERATE label_a;
        END IF;
        SET result = CAST(CONCAT(result, c) AS VARCHAR);
    END WHILE;
END
```

**LEAVE**

The `LEAVE` statement is used to abort execution of the current block.

**Syntax**

```
LEAVE <label>
```

**Remark**

- The `LEAVE` statement is equivalent to using `break` in Java. It aborts the current loop or compound statement block in an orderly way (this is not an error).
Example

-- Pads s with padChar so that s has at least width length.
PROCEDURE padr (IN s VARCHAR, IN width INTEGER, IN padChar VARCHAR,
OUT result VARCHAR)
L-padr:
BEGIN
  -- Returns null if any parameter is null
  IF s IS NULL OR width IS NULL OR padChar IS NULL THEN
    LEAVE L-padr;
  END IF;
...
END

LOOP

The LOOP statement is used for looping through the current block.

Syntax

[<label>:] LOOP
  <statements>
END LOOP [<label>]

This statement loops forever. You have to use a LEAVE statement to exit it.

Remarks

- The label is an optional identifier to name the block. This is for use with the
  LEAVE and ITERATE statements. See "LEAVE" on page 211 and "ITERATE" on page
  210.
- If a beginning label is present, the end label is not required. If no beginning
  label is present, then it is illegal to have an end label. If both the beginning and
  end labels are present, then both must have the same identifier.
- There may be zero or more statements in the <statements> area.
Example

-- Pads s with padChar so that s has at least width length.
PROCEDURE padr (IN a VARCHAR, IN width INTEGER, IN padChar VARCHAR,
OUT result VARCHAR)
-- pad result with padChar
  SET result = s;
L-loop:
  LOOP
  IF LENGTH(result) >= width THEN
    LEAVE L_loop;
  END IF;
  SET result = CAST(CONCAT(result, padChar) AS VARCHAR);
  END LOOP;
END

OPEN

The OPEN statement is used to open a cursor. Two types of OPEN statements are available, one to open a static cursor and another to open a variable cursor. The OPEN statement for a variable cursor can specify whether it is for a query expression or a value expression. See “Value Expressions” on page 154.

Syntax - Open Static Cursor

OPEN <cursor>

This statement works on both static and variable cursors, although you will get an error if you open an uninitialized cursor variable.

Syntax - Open Variable Cursor

OPEN <cursorVariableName> FOR <queryExpression>

A cursor variable can be opened and initialized using a dynamic SQL statement as follows:

OPEN <cursorVariableName> FOR <valueExpression>
Remarks
- Opening a cursor is the equivalent of executing the statement in JDBC. It prepares the result set for reading.
- It is illegal to open a cursor that is already open.

Errors
Standard parser and resolver errors may result from the `SELECT` statement in the `FOR` clause.

The following table describes the errors that may occur when executing an `OPEN` statement.

Table 59. Error Messages for an OPEN Statement

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot open a PIPE</td>
<td>An attempt is made to open a PIPE variable.</td>
</tr>
<tr>
<td>Uninitialized cursor</td>
<td>A cursor variable is used and is not initialized at the time it is opened.</td>
</tr>
<tr>
<td>Cursor already open</td>
<td>A cursor is opened when it is already open.</td>
</tr>
</tbody>
</table>

PATH
You can define paths to resources by providing a unique names to each path. `PATH` is similar to `import` in Java.

Remarks
- `PATH` should be specified in the first `BEGIN/END` as the first statement after `BEGIN`.
- Wherever you can use a variable, you can use `PATH`.
- `PATH` can be used to fully qualify unqualified tables or procedures used in the `FROM` clause, and `CALL` and `INSERT/DELETE/UPDATE` statements.

Syntax
`PATH <full path>`
Example

PROCEDURE p_path1(out outgoing int)
BEGIN
    PATH /users/composite/test/views;
    DECLARE public x constant int default 0;
    DECLARE public y constant int default 5;
    DECLARE public z constant int default 0;
    DECLARE public e1 exception;
    SET outgoing = y;
    EXCEPTION
        WHEN /users/composite/test/views/p_path1.e1 THEN
            RAISE
END

RAISE

The RAISE statement is used to raise an exception.

Syntax

RAISE [exceptionName] [VALUE [valueExpression]]

The value expression must resolve to a string. See "Value Expressions" on page 154.

Remarks

- The name, indicated by <exceptionName> in the syntax, can be any exception that is defined in the current scope, a parent scope, or that has a qualified name (such as a system exception).
- A name is required if this statement is outside of an exception handler. When inside an exception handler and when no name is used, the current exception is re-raised.
- The value, indicated by <valueExpression> in the syntax, can optionally be set on an exception. If not present, the value defaults to NULL. The value will be implicitly cast (if necessary) to be assigned into the exception.
- You can change the value of an exception when re-raising it by including the VALUE clause but no exception name.
**Examples**

PROCEDURE square (IN x INTEGER)
BEGIN
    DECLARE illegal_parameter_ex EXCEPTION;

    IF x IS NULL THEN
        RAISE illegal_parameter_ex;
    END IF;

    ...
END

PROCEDURE p (IN x INTEGER)
BEGIN
    DECLARE illegal_parameter_ex EXCEPTION;

    IF x < 0 THEN
        RAISE illegal_parameter_ex VALUE 'x must be > 0. x='||x;
    END IF;

    ...
END
REPEAT

The `REPEAT` statement is used to repeat specific statements under specific conditions.

Syntax

```sql
[label:] REPEAT
<statements>
UNTIL <conditionalExpression>
END REPEAT [label]
```

Remarks

- The label is an optional identifier to name the block. The `REPEAT` statement is for use with the `LEAVE` and `ITERATE` statements. See “LEAVE” on page 211 and “ITERATE” on page 210.
- If a beginning label is present, the end label is not required. If no beginning label is present, then it is illegal to have an end label. If both the beginning and end labels are present, then both must have the same identifier.
- There may be zero or more statements in the `<statements>` area.

Example

```sql
--Returns the root of id
PROCEDURE
BEGIN
   DECLARE parent_id INTEGER DEFAULT id;
   REPEAT
      SET result = parent_id;
      CALL /shared/parent_of(result, parent_id);
   UNTIL parent_id IS NULL
   END REPEAT;
END
```
ROLLBACK

If you are inside a compound statement with an independent transaction, you can
roll back the transaction. See “Compound Statement” on page 167 for details on
compound statements.

**Syntax**

```
ROLLBACK
```

**Remark**

• It is illegal to call ROLLBACK in a compound statement that is not declared
  independent.

**Example**

```
PROCEDURE p ( )
BEGIN INDEPENDENT TRANSACTION
  INSERT INTO /shared/T (name, score) VALUES ('Joe', 123);
  ROLLBACK;
END
```

SELECT INTO

Any SELECT statement that the system accepts can be used as a standalone SQL
Script statement as long as it uses the SELECT INTO format. A standalone SELECT
statement without the INTO clause will be discarded by the optimizer since it
would do nothing to the program state so it is disallowed.

**Syntax**

```
SELECT <projections> INTO <varListOrRowVariable>
FROM ...
```

Variables are allowed in a SQL statement anywhere a literal of the same type is
allowed.

**Remarks**

• The BOOLEAN and ROW types are not supported in SQL.
There is no special syntax for noting that something is a variable instead of a column in SQL statements, so be cautious when declaring a variable’s name. If there is a conflict, the name is interpreted as a column name and not a variable name.

- When using `SELECT INTO`, the cursor must return a single row. If it returns no rows, an exception will be raised. If it returns more than one row, an exception will be raised.

- Use of `SELECT INTO` is sometimes called an “implicit cursor” because it is opened, fetches one row, and is closed in one statement.

**Example**

```sql
PROCEDURE selinto_ex ( )
BEGIN
  DECLARE a INTEGER;
  DECLARE b DATE;
  SELECT col1, col2 INTO a, b FROM T WHERE x = 1;
END
```

**SET**

SET is an assignment statement, which assigns a value to a variable.

**Syntax**

```sql
SET <varName> = <value>
```

**Remarks**

- Values are coerced (implicitly cast) if that is possible.
- ROW values can be assigned to ROW variables only if each of the fields in the ROW variable could be assigned independently. Fields will be coerced (implicitly cast) as required.
- A cursor variable with a type can be assigned from any cursor with the same ROW type, or to any cursor variable with exactly the same ROW type.
A cursor variable without a type can be assigned from any cursor, or to any
cursor. Assigning to a typed cursor forces a runtime schema match comparison
and raises an exception on a mismatch.

Assigning a cursor creates a reference to the original cursor’s state. This
means that opening, closing, or fetching from the original cursor or the
variable has the same effect and alters what the other would see. See “OPEN”
on page 213, “CLOSE” on page 182, and “FETCH” on page 204 for details on
opening, closing, and fetching actions on cursors.

Errors
The following table describes the errors that may occur when executing a SET
statement.

Table 60. Error Messages for a SET Statement

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannot alter the value of an IN parameter</td>
<td>The specified variable is an IN parameter.</td>
</tr>
</tbody>
</table>

UPDATE

Any UPDATE statement that the system accepts can be used as a standalone SQL
Script statement.

Variables are allowed in a SQL statement anywhere a literal is allowed.

Syntax

```
UPDATE <table>
SET <column> = <valueExpression> [, <column> = <valueExpression>]*
[WHERE <conditionalExpression>]
```

The WHERE clause in the syntax is optional. The rules for the WHERE clause of an
UPDATE statement is the same as the rules for WHERE clause of a SELECT statement.

Remark

- Sub-queries in the SET clause, such as
UPDATE <table1> SET x = (SELECT y FROM <table2>)
are not permitted.

Examples
PROCEDURE p ( )
BEGIN
  DELETE FROM /shared/scores;
  INSERT INTO /shared/scores VALUES ('Joe', 1001);
  UPDATE /shared/.scores SET score=1239 WHERE name='Sue';
END

PROCEDURE p (IN p_name VARCHAR, IN new_score)
BEGIN
  DELETE FROM /shared/scores WHERE name=p_name;
  INSERT INTO /shared/scores VALUES (p_name, new_score);
  UPDATE /shared/.scores SET score=new_score WHERE name=p_name;
END

WHILE

The WHILE statement is used to execute certain statements as long as specific conditions are met.

Syntax
[<label>::] WHILE <conditionalExpression> DO
  <statements>
END WHILE [<label>]

The label, indicated by <label> in the syntax, is an optional identifier to name the block.

Remarks
• The WHILE statement is for use with the LEAVE and ITERATE statements. See “LEAVE” on page 211 and “ITERATE” on page 210.
• If a beginning label is present, the end label is not required. If no beginning label is present, then it is illegal to have an end label. If both the beginning and end labels are present, then both must have the same identifier.
• There may be zero or more statements in the `<statements>` area.

Examples

This section contains several examples illustrating the use of the SQL Script language. All the examples assumes a user named test in the domain composite.

Example 1: fetchExample1

This script iterates through a table and fetches all the rows. It assumes a Northwind access database named access and gathers all the categories in the table Categories.

Script

```sql
PROCEDURE fetchExample1 (OUT category CHAR)
BEGIN
    DECLARE temp CHAR;
    DECLARE f CURSOR FOR SELECT Categories.CategoryName FROM /shared/access/Categories Categories;

    SET category = '
;
    OPEN f;
    FETCH f INTO temp;
    -- Must call FETCH first, otherwise FOUND will be false.
    WHILE LOOP:
        WHILE f.FOUND DO
            BEGIN
                SET category = CAST(CONCAT(CONCAT(category, ' '), temp)AS CHAR(255));
                FETCH f INTO temp;
            END;
        END WHILE;
END;
END WHILE;
```
Examples Chapter 2 Composite SQL Script

CLOSE f;
END

Example 2: fetchExample2

This example is similar to Example 1: fetchExample1, but it fetches all the categories.

Script

PROCEDURE fetchExample2 (OUT category CHAR)
BEGIN
  DECLARE temp CHAR DEFAULT '';
  SET category = '';
  FOR x as SELECT Categories.CategoryName
  FROM /shared/access/Categories Categories
  DO
    SET temp = x.categoryName;
    SET category = CAST(CONCAT(CONCAT(category, ' '), temp) as CHAR);
  END FOR;
END

Example 3: type_example1

This example declares a user-defined type named \texttt{udt}, and uses it in another user-defined type \texttt{b}.

Script

PROCEDURE type_example1 ()
BEGIN
  DECLARE PUBLIC TYPE udt INTEGER;
  DECLARE TYPE b ROW (a INTEGER, b udt, c VARCHAR(255));
END
Example 4: type_example2

Script

```sql
PROCEDURE type_example2 ()
BEGIN
    -- b is defined in "Example 3: type_example1" on page 223
    DECLARE test /shared/type_example1.b;

    SET test.a = 123;
    SET test.b = 345;
    SET test.c = 'hello';
END
```

Example 5: pipe_example2

This example inserts the categories from the Northwind database into a PIPE variable.

Script

```sql
PROCEDURE pipe_example2 (OUT param1 PIPE (col1 CHAR), IN param2 INT)
BEGIN
    FOR x as SELECT Categories.CategoryName, Categories.CategoryId
    FROM /shared/access/Categories Categories
    DO
        IF x.CategoryId = param2 THEN
            INSERT INTO param1 (col1) VALUES (x.categoryName);
        END IF;
    END FOR;
    CLOSE param1;
END
```
Example 6: dynamic_sql_example

This example extracts data from a SELECT statement and uses an INSERT statement with the data. It extracts the values and inserts the values one by one.

Script

```sql
PROCEDURE dynamic_sql_example ()
BEGIN
  DECLARE sqltext VARCHAR DEFAULT 'INSERT INTO /shared/updates(c_varchar) VALUES('';
  DECLARE temp VARCHAR;
  FOR x AS SELECT Categories.CategoryName
    FROM /shared/access/Categories Categories
  DO
    SET temp = CAST(sqltext || x.CategoryName ||''')' as VARCHAR);
    EXECUTE IMMEDIATE temp;
  END FOR;
END
```

Example 7: dynamic_sql_example2

This example creates a dynamic SQL string to insert data from a variable. Instead of extracting the values, call the value by variable name.

Script

```sql
PROCEDURE dynamic_sql_example2 ()
BEGIN
  DECLARE sql2 VARCHAR DEFAULT 'INSERT INTO /shared/updates(c_varchar) VALUES('';
  DECLARE temp CHAR;
  FORLOOP:
    FOR x AS SELECT Categories.CategoryName
      FROM /shared/access/Categories Categories
    DO
      SET temp = CAST(sql2 || x.categoryName ||''')' as VARCHAR);
      EXECUTE IMMEDIATE temp;
    END FOR;
  END
```
Example 8: prepackaged_query_example

This example calls a prepackaged query, and returns the first row of data. It assumes that the user has a prepackaged query named pqAccess under the shared folder.

Script

PROCEDURE prepackaged_query_example ()
BEGIN
  -- Declare a cursor to retrieve from the prepackaged query
  DECLARE myRow ROW(a1 INT, a2 VARCHAR, a3 VARCHAR, a4 DECIMAL, a5 INT, a6 DECIMAL, a7 VARCHAR, a8 VARCHAR);
  DECLARE crs cursor(a1 int, a2 VARCHAR, a3 VARCHAR, a4 DECIMAL, a5 INT, a6 DECIMAL, a7 VARCHAR, a8 VARCHAR);
  CALL /shared/pqAccess(crs);
  -- Fetch the first row
  FETCH crs INTO myRow;
END

Example 9: exception_example

This example shows how to raise EXCEPTION.

Script

PROCEDURE exception_example (OUT has_error INT)
BEGIN
  DECLARE too_many_categories EXCEPTION;
  DECLARE no_categories EXCEPTION;
  DECLARE category_count INT DEFAULT 0;

SELECT COUNT(Categories.CategoryName) INTO category_count
FROM /shared/access/Categories Categories;
IF category_count > 5 THEN
  RAISE too_many_categories;
ELSEIF category_count = 0 THEN
  RAISE no_categories;
END IF;
SET has_error = 0;
EXCEPTION
  WHEN too_many_categories OR no_categories THEN
    SET has_error = 1;
END

Example 10: row_example

This example shows how to declare ROW

Script

PROCEDURE row_example()
BEGIN
  DECLARE category_row ROW (categoryid INT, categoryName CHAR);
  DECLARE f CURSOR FOR SELECT Categories.CategoryId,
    Categories.CategoryName
  FROM /shared/access/Categories Categories;

  OPEN f;
  FETCH f INTO category_row;
  CLOSE f;
  END
Example 11: divide

This example prevents "divide by zero" errors.

Script

```sql
PROCEDURE divide
(IN dividend INT, IN divisor INT, OUT result INT, OUT message CHAR)
BEGIN
    DECLARE divide_by_zero EXCEPTION;
    IF divisor = 0 THEN
        RAISE divide_by_zero value 'divided by zero error';
    END IF;
    SET result = dividend/divisor;
EXCEPTION
    WHEN divide_by_zero THEN
        SET message = CURRENT_EXCEPTION.MESSAGE;
    END
```
Chapter 3

Composite Query Engine Options

Execution of SQL views, procedures, and transactions created with CIS defined resources uses an optimized execution plan. The execution plan is generated dynamically based upon how the SQL is written, what and how native resources are being utilized, CIS configuration settings, the presence of data source specific statistical data, and any Composite SQL query engine options that direct how the execution plan is to be generated. This chapter describes the Composite SQL query engine options used to direct how the execution plan should be generated.

The query engine options enable the developer to override CIS settings for that specific SQL. Options may be applied to specific SQL keywords to influence the generation of the execution plan.

<table>
<thead>
<tr>
<th>SELECT Options</th>
<th>JOIN Options</th>
<th>UNION / INTERSECT / EXCEPT Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE_SENSITIVE</td>
<td>NESTEDLOOP</td>
<td>PARALLEL</td>
</tr>
<tr>
<td>IGNORE_TRAILING_SPACES</td>
<td>HASH</td>
<td>FORCE_DISK</td>
</tr>
<tr>
<td>DISABLE_DATA_CACHE</td>
<td>SORTMERGE</td>
<td>DISABLE_THREADS</td>
</tr>
<tr>
<td>DISABLE_STATISTICS</td>
<td>SHMJOIN</td>
<td>DISABLE_PLANCACHE</td>
</tr>
<tr>
<td>DISABLE_CBO</td>
<td>FORCE_ORDER</td>
<td>DISABLE_PUSH</td>
</tr>
<tr>
<td>MAX_ROWS_LIMIT</td>
<td>SNAP_ORDER</td>
<td>STRICT</td>
</tr>
<tr>
<td>FORCE_DISK</td>
<td>LEFT_CARDINALITY</td>
<td></td>
</tr>
<tr>
<td>DISABLE_THREADS</td>
<td>RIGHT_CARDINALITY</td>
<td></td>
</tr>
<tr>
<td>DISABLE_PLANCACHE</td>
<td>FORCE_DISK</td>
<td></td>
</tr>
<tr>
<td>DISABLE_PUSH</td>
<td>DISABLE_THREADS</td>
<td></td>
</tr>
<tr>
<td>STRICT</td>
<td>DISABLE_PUSH</td>
<td></td>
</tr>
<tr>
<td>INSERT / UPDATE / DELETE Options</td>
<td>FORCE_DISK</td>
<td>PARTITION_SIZE</td>
</tr>
<tr>
<td>CASE_SENSITIVE</td>
<td>DISABLE_THREADS</td>
<td></td>
</tr>
<tr>
<td>IGNORE_TRAILING_SPACES</td>
<td>DISABLE_PUSH</td>
<td></td>
</tr>
<tr>
<td>STRICT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SELECT Options

SELECT options are specified immediately following the SELECT keyword.

Example:

```
SELECT {OPTION FORCE_DISK}
SELECT {OPTION FORCE_DISK="TRUE"}
SELECT {OPTION FORCE_DISK="FALSE"}
```

If an option is specified without a value, the default value is set to "true".

For example: {OPTION FORCE_DISK} is equivalent to {OPTION FORCE_DISK="TRUE"}

Operators (such as JOIN) level options override SELECT level options.

Please note that any option specified in a SELECT other than the root-level one may
or may not take effect in an unpredictable way. Hence, SELECT level options
should not be used in any saved views or in sub-selects.

CASE_SENSITIVE

If true, sets the string comparisons in a case sensitive mode. This option overrides
the Composite Server’s default case sensitivity setting.

If this option is not specified, Composite Server’s case sensitivity setting will
determine how the string comparisons are evaluated.

Usage:

```
SELECT {OPTION CASE_SENSITIVE="TRUE"} * FROM table1
WHERE column1 = 'FOO'
```
**IGNORE_TRAILING_SPACES**

If true, sets the comparisons to ignore trailing spaces. This option overrides the Composite Server’s ignore trailing spaces setting.

If this option is not specified, Composite Server’s ignore trailing spaces setting will determine how the string comparisons are evaluated.

Usage:

```sql
SELECT {OPTION IGNORE_TRAILING_SPACES="FALSE"} * FROM table1
WHERE column1 = 'FOO'
```

**DISABLE_DATA_CACHE**

If true, the query will be executed as if the cached views used in the query are not cached. This option can be useful for certain queries that require the latest data, not the cached data.

If this option is not specified, the data from the cache will always be used for all cached views.

Usage:

```sql
SELECT {OPTION DISABLE_DATA_CACHE} * FROM cachedView1
```

**DISABLE_STATISTICS**

If true, the query planning will be done as if the statistics were not collected on any of the tables referenced in the query. This option can be useful to compare how gathering statistics improves the query plan.

If this option is not specified, query planning will use any statistics that is available to generate a better query plan.

Usage:

```sql
SELECT {OPTION DISABLE_STATISTICS} * FROM table1
WHERE column1 = 5
```
**DISABLE_CBO**

Disabling the cost-based optimizations (CBO) will force the execution plan to be generated from rule-based heuristics. When set to true, the query optimizer will ignore any table boundary or other table statistics that may have been gathered and it will optimize the execution plan based only on the heuristics, or rule-based optimization.

If this option is not specified, the query optimizer will apply cost based optimizations in addition to heuristics based optimizations.

Usage:

```sql
SELECT {OPTION DISABLE_CBO} * FROM table1 INNER JOIN table2 ON table1.id = table2.id
```

**MAX_ROWS_LIMIT**

This option can be used to limit the maximum number of rows returned by a query. This is useful if a user is interested only in the first 'n' rows of the query.

If this option is not specified, all the rows will be returned.

Usage:

```sql
SELECT {OPTION MAX_ROWS_LIMIT=100} * FROM table1
```

**FORCE_DISK**

If true, forces Query Engine to use disk instead of memory for temporary storage of data that is required to process the query. This frees up memory for other server operations and is particularly useful for certain queries that consume lot of memory and affect performance of all the other running queries in the server.

If this option is not specified, Query Engine will try to use memory instead of disk whenever possible for maximum performance.

Usage:

```sql
SELECT {OPTION FORCE_DISK} * FROM table1 INNER JOIN table2 ON table1.id = table2.id INNER JOIN table3 ON table1.id = table3.id
```
DISABLE_THREADS
If true, Query Engine will not use background threads to speed up processing. This option can be useful to force certain resource intensive queries to not use Server’s resources preemptively.
If this option is not specified, Query Engine will always use background threads to speed up processing.
Usage:
```
SELECT {OPTION DISABLE_THREADS} * FROM table1 INNER JOIN
table2 ON table1.id = table2.id INNER JOIN table3 ON
table1.id = table3.id
```

DISABLE_PLAN_CACHE
If true, Query Engine will prepare a fresh query plan every time for executing the query.
If this option is not specified, Query Engine will always use a cached plan if it is available in the cache.
Usage:
```
SELECT {OPTION DISABLE_PLAN_CACHE} * FROM table1
```

DISABLE_PUSH
If true, the entire SELECT will be processed locally in Composite Server instead of pushing it to the data source.
If this option is not specified, the entire SELECT will always be pushed to the data source if possible.
Usage:
```
SELECT {OPTION DISABLE_PUSH} column1 FROM table1 INNER JOIN
table2 ON table1.id = Table2.id
```
**STRICT**

If true, we don’t push certain aspects of SQL (such as mathematical functions) to adhere to strict SQL 92 behavior. This could affect performance.

If this option is not specified, we relax some SQL 92 rules to achieve more push.

Usage:

```
SELECT [OPTION STRICT] tan(colum1) FROM table1
```
**FORCE_DISK**

If true, disk will be used instead of memory for temporary storage of data that is required to process the **UNION**, **INTERSECT**, or **EXCEPT** operators. This frees up memory for other server operations and is particularly useful for certain queries that consume lot of memory and affect performance of all the other running queries in the server.

If this option is not specified, memory will be used instead of disk whenever possible for maximum performance.

This has higher precedence over the **SELECT** level **FORCE_DISK** option.

Usage:

```
SELECT column1 FROM table1
UNION ALL {OPTION FORCE_DISK}
SELECT column1 FROM table2
```

**DISABLE_PUSH**

If true, **UNION**, **INTERSECT**, and **EXCEPT** operators will be processed locally in Composite Server instead of pushing it to the data source.

If this option is not specified, **UNION**, **INTERSECT**, and **EXCEPT** operators will always be pushed to the data source if possible.

Usage:

```
SELECT column1 FROM table1
UNION ALL {OPTION DISABLE_PUSH}
SELECT column1 FROM table2
```
JOIN Options

Join options are specified using SQL 92 join syntax.

NESTEDLOOP

If true, optimizer will choose Nested loop join algorithm for the join. If false, the optimizer will not consider Nested loop join algorithm, if the join can be evaluated using other join algorithms.

If this option is not specified, the optimizer decides the best algorithm for the join.

Usage:

```sql
SELECT column1 FROM table1 INNER [OPTION NESTEDLOOP] JOIN table2 ON table1.id = table2.id
```

HASH

If true, optimizer will attempt to choose Hash algorithm if possible. If false, the optimizer will not consider Hash algorithm when it tries to decide the best algorithm for evaluating the join.

If this option is not specified, the optimizer decides the best algorithm for the join.

Usage:

```sql
SELECT column1 FROM table1 INNER [OPTION HASH] JOIN table2 ON table1.id = table2.id
```

SORTMERGE

If true, optimizer will attempt to choose the Sort Merge algorithm if possible. If false, the optimizer will not consider Sort Merge algorithm when it tries to decide the best algorithm for evaluating the join.

If this option is not specified, the optimizer decides the best algorithm for the join.

Usage:

```sql
SELECT column1 FROM table1 INNER [OPTION SORTMERGE] JOIN table2 ON table1.id = table2.id
```
SEMIJOIN

With this option query engine hint the optimizer will attempt to perform a semi join optimization. SEMIJOIN is a very fast algorithm that reduces the number of rows retrieved from the RHS by rewriting the FETCH pushed to the second data source with selective criteria provided by the unique values returned from an initial query on the LHS. While the other join algorithms can be found in traditional database products, the semi-join is exclusively an Information Integration tool. In the semi-join, the left side is evaluated and loaded into an in-memory table. Then the cardinality is evaluated. If the cardinality is small enough, an IN clause or an OR expression is created containing all the values in the join criteria from the left side. That is then appended to the WHERE clause on the right hand side and pushed to the database. In this way, only rows which will have matches are retrieved from the right side.

If this option is not specified the optimizer decides whether to apply semi join optimization or not.

Usage:

```
SELECT column1 FROM table1 INNER {OPTION SEMIJOIN} JOIN table2 ON table1.id = table2.id
```

Note: The semi-join can only be attempted if the right hand side may be queried as a single node which fetches against a data source that supports IN or an OR clause.

FORCE_ORDER

If true, the optimizer will honor the order of the joins specified in the sql.

If this option is not specified, the optimizer may switch the order of joins if it will result in a better query plan.

Usage:

```
SELECT column1 FROM table1 INNER {OPTION FORCE_ORDER} JOIN table2 ON table1.id = table2.id
```
SWAP_ORDER
If true, swaps the order of the join after the SQL is parsed. This can be useful for queries with complex joins where it is easier to use this option to swap the join order than trying to move lots of text around in the SQL.
If this option is not specified, we will use the parsed join order.
Usage:
```sql
SELECT column1 FROM table1 INNER {option swap_order} JOIN table2 ON table1.id = table2.id
```

LEFT_CARDINALITY
This option provides cardinality hint for the left hand side of a join. The hint will be used by the optimizer to choose a better query plan.
If this option is not specified, the optimizer will rely on statistics processing for cardinality estimates.
Usage:
```sql
SELECT column1 FROM table1 INNER {OPTION LEFT_CARDINALITY=10} JOIN table2 ON table1.id = table2.id
```

RIGHT_CARDINALITY
This option provides cardinality hint for the right hand side of a join. The hint will be used by the optimizer to choose a better query plan.
If this option is not specified, the optimizer will rely on statistics processing for cardinality estimates.
Usage:
```sql
SELECT column1 FROM table1 INNER {OPTION RIGHT_CARDINALITY=10000} JOIN table2 ON table1.id = table2.id
```
FORCE_DISK
If true, disk will be used instead of memory for temporary storage of data that is required to process the join operator. This frees up memory for other server operations and is particularly useful for certain queries that consume lot of memory and affect performance of all the other running queries in the server.
If this option is not specified, memory will be used instead of disk whenever possible in an attempt to maximize performance.
This has higher precedence over the SELECT level FORCE_DISK option.
Usage:
```sql
SELECT column1 FROM table1 INNER {OPTION FORCE_DISK} JOIN table2 ON table1.id = table2.id
```

DISABLE_THREADS
If true, Query Engine will not use background threads to speed up processing. This option can be useful to force certain resource intensive queries to not use Server's resources preemptively.
If this option is not specified, Query Engine will always use background threads to speed up processing.
This has higher precedence over the SELECT level DISABLE_THREADS option.
Usage:
```sql
SELECT column1 FROM table1 INNER {OPTION DISABLE_THREADS} JOIN table2 ON table1.id = table2.id SELECT column1 FROM table2
```

DISABLE_PUSH
If true, the join operator will be processed locally in Composite Server instead of pushing it to the data source.
JOIN Options

If this option is not specified, the join operator will always be pushed to the data source if possible.

Usage:

```sql
SELECT column1 FROM table1 INNER {OPTION DISABLE_PUS} JOIN table2 ON table1.id = table2.id
```

PARTITION_SIZE

The partition_size join option may be used to manually restrict the size of the condition clause submitted to the RHS of a semijoin by specifying the maximum number of condition arguments that may be sent in a each batch request.

This may be advantageous if a large cardinality result set is expected from the LHS of a semijoin and the RHS SQL select statement must be limited in size because of data resource limitations (i.e., limited acceptable SQL string length).

To limit the partition size sent to the RHS, set the partition_size option to an integer representing the number of arguments in the condition clause submitted to the second datasource.

Usage example:

```sql
SELECT TableX.col1 FROM /Folder/SomeResource/DatabaseX TableX INNER {OPTION PARTITION_SIZE=9} JOIN /FolderY/ResourceZ TableY.col2 ON TableX.oid = TableY.oid
```
GROUP BY Options

FORCE_DISK

If true, disk will be used instead of memory for temporary storage of data that is required to process the group by operator. This frees up memory for other server operations and is particularly useful for certain queries that consume lot of memory and affect performance of all the other running queries in the server.

If this option is not specified, memory will be used instead of disk whenever possible for maximum performance.

This has higher precedence over the SELECT level FORCE_DISK option.

Usage:
```
SELECT MAX(column2) FROM table1
GROUP BY {OPTION FORCE_DISK} column1
```

DISABLE_THREADS

If true, Query Engine will not use background threads to speed up processing of the group by operator. This option can be useful to force certain resource intensive queries to not use Server’s resources preemptively.

If this option is not specified, Query Engine will always use background threads to speed up processing.

This has higher precedence over the SELECT level DISABLE_THREADS option.

Usage:
```
SELECT MAX(column2) FROM table1
GROUP BY {OPTION DISABLE_THREADS} column1
```
**DISABLE_PUSH**

If true, the group by operator will be processed locally in Composite Server instead of pushing it to the data source.

If this option is not specified, the group by operator will always be pushed to the data source if possible.

Usage:

```
SELECT MAX(column2) FROM table1
GROUP BY {OPTION DISABLE_PUSH} column1
```

**ORDER BY Options**

**FORCE_DISK**

If true, disk will be used instead of memory for temporary storage of data that is required to process the order by operator. This frees up memory for other server operations and is particularly useful for certain queries that consume lot of memory and affect performance of all the other running queries in the server.

If this option is not specified, memory will be used instead of disk whenever possible for maximum performance.

This has higher precedence over the SELECT level FORCE_DISK option.

Usage:

```
SELECT column1 FROM table1
ORDER BY {OPTION FORCE_DISK} column1
```
DISABLE_THREADS

If true, Query Engine will not use background threads to speed up processing of the order by operator. This option can be useful to force certain resource intensive queries to not use Server’s resources preemptively.

If this option is not specified, Query Engine will always use background threads to speed up processing.

This has higher precedence over the SELECT level DISABLE_THREADS option.

Usage:

```
SELECT column1 FROM table1
ORDER BY [OPTION DISABLE_THREADS] column1
```

DISABLE_PUSH

If true, the ORDER BY operator will be processed locally in Composite Server instead of pushing it to the data source.

If this option is not specified, the order by operator will always be pushed to the data source if possible.

Usage:

```
SELECT column1 FROM table1
ORDER BY [OPTION DISABLE_PUSH] column1
```
INSERT / UPDATE / DELETE Options

Insert, update, and delete options are specified right after the `INSERT`, `UPDATE` and `DELETE` keywords respectively.

**CASE_SENSEITIVE**

If true, sets the string comparisons in a case sensitive mode. This option overrides the Composite Server’s default case sensitivity setting.

If this option is not specified, Composite Server’s case sensitivity setting will determine how the string comparisons are evaluated.

Usage:

```
UPDATE {OPTION CASE_SENSITIVE="TRUE"} table1
SET column1 = 'BAR'
WHERE column1 = 'FOO'
```

**IGNORE_TRAILING_SPACES**

If true, sets the comparisons to ignore trailing spaces. This option overrides the Composite Server’s “ignore trailing spaces” setting.

If this option is not specified, Composite Server’s ignore trailing spaces setting will determine how the string comparisons are evaluated.

Usage:

```
UPDATE {OPTION IGNORE_TRAILING_SPACES="FALSE"} table1
SET column1 = 'BAR'
WHERE column1 = 'FOO'
```
**STRICT**

If `{option strict}` is specified, we don’t push certain aspects of SQL (such as mathematical functions) to adhere to strict SQL 92 behavior. This could affect performance.

If this option is not specified, we relax some SQL 92 rules to achieve more push.

Usage:

```
UPDATE {OPTION STRICT} table1
SET column2 = 'S'
WHERE SIN(column1) = 1
```

**CHECK_VIEW_CONSTRAINTS**

If false, Composite Server does not preserve the data integrity of the view definition. If true, Composite Server preserves the data integrity of the view definition and disallows changes to the view.

If this option is not specified, Composite Server always preserves the data integrity of the view definition.

For example,

Suppose a view V1 is defined as follows:

```
SELECT column1 FROM table1 WHERE column1 = 5
```

If someone tries to update V1 with the following update statement

```
UPDATE V1
SET column1 = 5
WHERE column1 = 6
```

The update statement will fail if this option is set to true because row with value `column1=6` is outside the bounds of the definition of the view V1.
Chapter 4

Composite Built-in Procedures

Composite provides a standard procedure library, similar to such an utility in the Oracle database.

The built-in procedures extend the Composite SQL Script language, much like Java is extended through classes. These procedures function exactly like Composite’s custom Java procedures.

Currently, the following built-in procedures are available. They are divided into the following groups—debug, resource, services, users, and util—and are available at the system level. You can call the built-in procedures from any other procedure. You can also publish them as Composite data services and call them from client applications.

<server-host>/lib/debug/
  Log
  LogError
  Print

<server-host>/lib/resource/
  CancelResourceReintrospect
  CancelResourceStatistics
  ClearResourceCache
  CopyResource
  CreateResourceCacheKey
  GetDataSourceReintrospectReport
  GetResourceStatisticsReport
  LoadResourceCacheStatus
  MoveResource
  RefreshResourceCache
Chapter 4 Composite Built-in Procedures

<server-host>/lib/resource/ (continued)
- RefreshResourceStatistics
- ReintrospectDataSource
- RenameResource
- ResourceExists
- SendResultsInEmail
- TestAllDataSourceConnections
- TestDataSourceConnection
- UpdateResourceCacheEnabled
- UpdateResourceEnabled

<server-host>/lib/services/
- AddUsernameToken
- CreateElement
- DeleteElement
- EncryptElement
- LogMessageToFile
- ProcessSecurityHeader
- SetEnvironmentFromNodeValue
- SetNodeValueFromEnvironment
- SignElement

<server-host>/lib/users/
- SyncDomain

<server-host>/lib/util
- GenerateEvent
- GetEnvironment
- GetProperty
- Pause
- SendEMail
- SetEnvironment
Notes
The path to a built-in procedure (/lib/debug/ or /lib/resource/ or
/lib/services/ or /lib/users/ or /lib/util/) is automatically added to every
script. This is similar to how Java imports java.lang.*. So there is no need to
fully qualify the built-in procedures when you call them from another procedure.
The section “Built-in Procedures” on page 248 describes the built-in procedures
in /lib/debug and /lib/util.

For additional details on those procedures and the procedures in
/lib/resource, /lib/services, and /lib/users, see the corresponding
Info panel in the Modeler.

User-Defined Procedures vs. Built-in Procedures
User-defined procedures with names that are identical to the name of any
Composite built-in procedure will result in a conflict. For example, if you write a
procedure with the name print or getProperty, the system will automatically fill
in the path to the built-in procedure (/lib/debug/print or
/lib/util/getProperty) rather than to your procedure even if you specify the
path to your procedure. This happens because the system path to the same-named
built-in procedure takes precedence.

Built-in Procedures
This section describes the built-in procedures in /lib/debug and /lib/util.

Log
This procedure writes the text you provide to the log file with severity level INFO.

Syntax
log (IN text VARCHAR (4096))
Example
PROCEDURE proc1()
BEGIN
    CALL Log('Hello');
    CALL Log('Hello World');
END

LogError
This procedure writes the text you provide to the log file with severity level ERROR.

Syntax
logError (IN text VARCHAR (4096))

Example
PROCEDURE proc2()
BEGIN
    CALL logError('Note that there is an error.');
END

Print
This procedure writes the text you provide to the Studio console. These prints are available for the specific script being run, and are not carried across scripts. The print messages are displayed in the Studio.

Syntax
print (IN text VARCHAR (4096))
Example

PROCEDURE proc3()
BEGIN
    CALL print('Printing to the Studio console...');
END

GenerateEvent

This procedure generates a custom event with the specified name and value. It can be used to activate a trigger that is configured to listen for this event name.

Syntax

generateEvent (IN eventName VARCHAR (40), IN value VARCHAR (4096))

Example

PROCEDURE CallsGenEv()
BEGIN
    CALL GenerateEvent('runAReport', ' ');
END

GetEnvironment

This procedure gets the environment variables, such as NUM_ROWS_AFFECTED, from the last operation.
An environment is applicable per-procedure, and is not global.

Syntax

getAddress (IN propName VARCHAR (40),
        OUT propValue VARCHAR (2048))
Note

Currently, the following environment variables are supported:

- System.CASE_SENSITIVE_IN_COMPARISONS
- System.IGNORE_TRAILING_SPACES_IN_COMPARISONS
- System.NUM_ROWS_AFFECTED
- System.TRIGGER_EVENT_NAME
- System.TRIGGER_EVENT_TYPE
- System.TRIGGER_EVENT_VALUE
- System.TRIGGER_PATH

Example

PROCEDURE proc4()
BEGIN
    PATH /shared/sources/scripts;
    DECLARE x VARCHAR(4096);
    CALL insertProc(); -- This procedure is in the PATH
    CALL getEnvironment('NUM_ROWS_AFFECTED', x);
    CALL log(x);
END

GetProperty

This procedure provides a way to get system properties such as current user ID, user name, and user domain. Properties are global and shared across scripts.

Syntax

getProperty {
    IN propName VARCHAR (40),
    OUT propValue VARCHAR (2048)}


Note
Currently, the following properties are supported.
  CURRENT_ID
  CURRENT_USER_ID
  CURRENT_USER_DOMAIN
  CURRENT_USER_NAME
  SERVER_HOSTNAME
  SERVER_ID
  SERVER_JDBC_PORT
  SERVER_VERSION
  SERVER_VERSION_NUMBER
  SERVER_WEB_PORT
  SESSION_ID
  TRANSACTION_ID

Example
PROCEDURE proc5()
BEGIN
  DECLARE x VARCHAR(4096);
  CALL getProperty('CURRENT_USER_ID', x);
  CALL log(x);
  CALL getProperty('CURRENT_USER_NAME', x);
  CALL log(x);
  CALL getProperty('CURRENT_USER_DOMAIN', x);
  CALL log(x);
END
Pause

This procedure provides a way to specify a sleep time (in milliseconds) for script execution.

**Syntax**

```plaintext
pause (IN msecs INTEGER)
```

**Example**

```plaintext
PROCEDURE proc6()
BEGIN
    CALL log('pausing for 3 secs');
    CALL pause(3000);
    CALL log('pause completed');
END
```

SendEMail

This procedure sends an e-mail message with the specified headers and content.

**Syntax**

```plaintext
SendEMail (IN from VARCHAR (4096),
            IN replyTo VARCHAR (4096),
            IN to VARCHAR (4096),
            IN cc VARCHAR (4096),
            IN bcc VARCHAR (4096),
            IN subject VARCHAR (4096),
            IN contentType VARCHAR (4096),
            IN content VARCHAR (4096))
```
Note
Current Release supports only NULL for the from address. NULL causes the use of the server’s configured from address.

Example
PROCEDURE proc_SendEMail()
BEGIN
  PATH /shared/sources/proceduresForDoc;
  CALL procGetProperty();
  CALL SendEMail(NULL, NULL, ’joe@smith.com’,
    NULL, NULL, ’hi’, ’TEXT_PLAIN’, NULL);
END

SetEnvironment
This procedure lets you set environment variables such as NUM_ROWS_AFFECTED from the last operation. Environment variable settings can generally be changed. This procedure lets you set a specific script’s NUM_ROWS_AFFECTED return value.

Syntax
setEnvironment (IN propName VARCHAR (40),
    IN propValue VARCHAR (2048))

Note
Currently, the following environment variables are supported:
  System.CASE_SENSITIVE
  System.IGNORE_TRAILING_SPACES
  System.NUM_ROWS_AFFECTED
  System.TRIGGER_EVENT_NAME
  System.TRIGGER_EVENT_TYPE
  System.TRIGGER_EVENT_VALUE
  System.TRIGGER_PATH
Example

PROCEDURE proc7()
BEGIN
    DECLARE x VARCHAR(4096);
    CALL getEnvironment('NUM_ROWS_AFFECTED', x);
    CALL log(x);
    SET x = '100';
    CALL setEnvironment('NUM_ROWS_AFFECTED', x);
    CALL getEnvironment('NUM_ROWS_AFFECTED', x);
    CALL log(x);
END
Chapter 5

Data Type Mappings

This chapter contains tables that map native data types used in different data sources to Composite JDBC data types.

Data types used in the following data sources are mapped to Composite JDBC data types:
- Oracle, on page 256
- SQL Server, on page 263
- DB2, on page 264
- Informix, on page 266
- Sybase, on page 267
- Teradata, on page 269
- MySQL, on page 270
- LDAP, on page 275
- CSV Flat File, on page 275
- MS Access, on page 276
- MS Excel, on page 277

Oracle Data Types and Composite JDBC Data Types

This section maps Oracle data types to Composite JDBC data types.

Oracle NUMBER data type and Composite JDBC data type

The following details apply to the mapping of Oracle NUMBER data type to Composite JDBC data type.
If the data type of an Oracle column is defined as `NUMBER`, the mapping works as follows:

- If the scale of the `NUMBER` column is not specified, it will be mapped as `DOUBLE`.
- If the either the scale or the precision is `NULL`, the data type will be mapped to `DOUBLE`.
- If the precision and scale are defined as non-zero values, the data type will be mapped to `DECIMAL`.
- If the scale is 0 (zero), different precision values will affect the data type mapping:
  - If the precision is less than and equal to 2, it is mapped to `TINYINT`.
  - If the precision is less than and equal to 4, it is mapped to `SMALLINT`.
  - If the precision is less than and equal to 9, it is mapped to `INTEGER`.
  - If the precision is less than and equal to 19, it is mapped to `BIGINT`.
  - Otherwise, it is mapped to `NUMERIC` with 0 (zero) scale.
- If the precision is not specified, it defaults to 38.

- When casting a value as `DECIMAL(p, s)`, as for example `CAST (Oracle_column AS DECIMAL(40))`, where the precision is greater than 38 (`p > 38`), it is processed in Composite. The maximum scale that Composite can support is 255. Any larger scale will be automatically reduced to 255. The maximum precision supported in Composite is `Integer.MAX_VALUE`, which is 2147483647.
## Oracle 8i Data Types and Composite JDBC Data Types

The following table maps Oracle 8i data types to Composite JDBC data types.

<table>
<thead>
<tr>
<th>Oracle 8i Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFILE</td>
<td>BLOB</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>LONG</td>
<td>CLOB</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>BLOB</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>NCLOB</td>
<td>CLOB</td>
</tr>
</tbody>
</table>
See also “Oracle NUMBER data type and Composite JDBC data type” on page 256.

In the following examples, the Oracle type is given on the left, and Composite JDBC type on the right. A hyphen rendered in bold face indicates that the value is not specified in Oracle.

<table>
<thead>
<tr>
<th>Oracle 8i Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>NUMBER 2 0</td>
<td>TINYINT</td>
</tr>
<tr>
<td>NUMBER 4 0</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>NUMBER 8 0</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NUMBER 15 0</td>
<td>BIGINT</td>
</tr>
<tr>
<td>NUMBER 22 0</td>
<td>NUMERIC(22,0)</td>
</tr>
<tr>
<td>NUMBER 10 3</td>
<td>DECIMAL(10,3)</td>
</tr>
<tr>
<td>NUMBER - 0</td>
<td>NUMERIC(38,0)</td>
</tr>
<tr>
<td>NUMBER - 2</td>
<td>DECIMAL(38,2)</td>
</tr>
<tr>
<td>NUMBER - -</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>RAW</td>
<td>VARBINARY(10)</td>
</tr>
<tr>
<td>BOMID</td>
<td>VARCHAR(10)</td>
</tr>
<tr>
<td>UROWID</td>
<td>VARCHAR(10)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>
## Oracle 9i Data Types and Composite JDBC Data Types

The following table maps Oracle 9i data types to Composite JDBC data types.

<table>
<thead>
<tr>
<th>Oracle 9i Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYDATA</td>
<td>OTHER</td>
</tr>
<tr>
<td>ANYDATASET</td>
<td>OTHER</td>
</tr>
<tr>
<td>ANYTYPE</td>
<td>OTHER</td>
</tr>
<tr>
<td>BFILE</td>
<td>BLOB</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR (n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>LONG</td>
<td>CLOB</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>BLOB</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR (n)</td>
</tr>
<tr>
<td>NCLOB</td>
<td>CLOB</td>
</tr>
</tbody>
</table>
### Table 62. Oracle 9i data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Oracle 9i Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>NUMBER 2 0</td>
<td>TINYINT</td>
</tr>
<tr>
<td>NUMBER 4 0</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>NUMBER 8 0</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NUMBER 15 0</td>
<td>BIGINT</td>
</tr>
<tr>
<td>NUMBER 22 0</td>
<td>NUMERIC(22,0)</td>
</tr>
<tr>
<td>NUMBER 10 3</td>
<td>DECIMAL(10,3)</td>
</tr>
<tr>
<td>NUMBER - 0</td>
<td>NUMERIC(38,0)</td>
</tr>
<tr>
<td>NUMBER - 2</td>
<td>DECIMAL(38,2)</td>
</tr>
<tr>
<td>NUMBER 12</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>NUMBER -</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>NVARCHAR2</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>RAW</td>
<td>VARBINARY(10)</td>
</tr>
<tr>
<td>ROWID</td>
<td>VARCHAR(10)</td>
</tr>
<tr>
<td>URTYPE</td>
<td>OTHER</td>
</tr>
<tr>
<td>UROWID</td>
<td>VARCHAR(10)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARCHAR2</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>XMLTYPE</td>
<td>XML</td>
</tr>
</tbody>
</table>

See also “Oracle NUMBER data type and Composite JDBC data type” on page 256.

In the following examples, the Oracle type is given on the left, and Composite JDBC type on the right. A hyphen rendered in bold face indicates that the value is not specified in Oracle.

- NUMBER 2 0 ---> TINYINT
- NUMBER 4 0 ---> SMALLINT
- NUMBER 8 0 ---> INTEGER
- NUMBER 15 0 ---> BIGINT
- NUMBER 22 0 ---> NUMERIC(22,0)
- NUMBER 10 3 ---> DECIMAL(10,3)
- NUMBER - 0 ---> NUMERIC(38,0)
- NUMBER - 2 ---> DECIMAL(38,2)
- NUMBER 12 ---> DOUBLE
- NUMBER - ---> DOUBLE
Oracle 10g Data Types and Composite JDBC Data Types

The following table maps Oracle 10g data types to Composite JDBC data types.

Table 63. Oracle 10g data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Oracle 10g Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYDATA</td>
<td>OTHER</td>
</tr>
<tr>
<td>ANYDATASET</td>
<td>OTHER</td>
</tr>
<tr>
<td>ANYTYPE</td>
<td>OTHER</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>Uses FLOOR( ) instead of ROUND( ) on the difference</td>
</tr>
<tr>
<td>TIMESTAMP(#), where # ranges from 0 to 9.</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP(#) with time zone, where # ranges from 0 to 9.</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP(#) with LOCAL time zone, where # ranges from 0 to 9.</td>
<td>OTHER</td>
</tr>
<tr>
<td>INTERVAL_YEAR(#) TO MONTH where # ranges from 0 to 9.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>INTERVAL_DAY(#) TO SECOND(#) where # ranges from 0 to 9.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>SDO_GEORASTER</td>
<td>OTHER</td>
</tr>
<tr>
<td>SI_STILLIMAGE</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>URI_TYPE</td>
<td>OTHER</td>
</tr>
<tr>
<td>UROWID</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>XMLTYPE</td>
<td>OTHER</td>
</tr>
</tbody>
</table>
### SQL Server Data Types and Composite JDBC Data Types

The following table maps SQL Server data types to Composite JDBC data types.

<table>
<thead>
<tr>
<th>SQL Server Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BINARY</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>BIT</td>
<td>BIT</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>IMAGE</td>
<td>BLOB</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>NTASK</td>
<td>CLOB</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC(p, s)</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>DECIMAL(p, s)</td>
</tr>
</tbody>
</table>
The following table maps DB2 (version 7.x and version 8.x) data types to Composite JDBC data types.

### Table 64. SQL Server data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>SQL Server Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL VARIANT</td>
<td>OTHER</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>VARBINARY(8)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>UNIQUEIDENTIFIER</td>
<td>CHAR(36)</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

### DB2 (v 7.x and v 8.x) Data Types and Composite JDBC Data Types

The following table maps DB2 (version 7.x and version 8.x) data types to Composite JDBC data types.

### Table 65. DB2 (v 7.x, and v 8.x) data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>DB2 (v 7.x and v 8.x) Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHARACTER_VARYING</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>DB2 (v 7.x and v 8.x) Data Type</td>
<td>Composite JDBC Data Type</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>CHAR_()_FOR_BIT_DATA</td>
<td>BINARY</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LONGVARGRAPHIC</td>
<td>CLOB</td>
</tr>
<tr>
<td>LONG_VARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>LONG_VARCHAR_()_FOR_BIT_DATA</td>
<td>BLOB</td>
</tr>
<tr>
<td>LONG_VARCHAR_()_FOR_BIT_DATA</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARGRAPH</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

Table 65. DB2 (v 7.x, and v 8.x) data types and Composite JDBC data types mapped
Informix 9.x Data Types and Composite JDBC Data Types

The following table maps Informix 9.x data types to Composite JDBC data types.

### Table 65. DB2 (v 7.x, and v 8.x) data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Composite JDBC Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARGRAPHIC</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

### Informix 9.x Data Types and Composite JDBC Data Types

The following table maps Informix 9.x data types to Composite JDBC data types.

### Table 66. Informix data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Informix Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BIT</td>
</tr>
<tr>
<td>BYTE</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DEC</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DOUBLE_PRECISION</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8</td>
<td>LONG</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>
The following table maps Sybase data types to Composite JDBC data types.

Table 66. Informix data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Informix Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>SERIAL</td>
<td>INTEGER</td>
</tr>
<tr>
<td>SERIAL8</td>
<td>LONG</td>
</tr>
<tr>
<td>SMALLFLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TEXT</td>
<td>CLOB</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>

Sybase Data Types and Composite JDBC Data Types

The following table maps Sybase data types to Composite JDBC data types.

Table 67. Sybase data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Sybase Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>BINARY (n)</td>
</tr>
<tr>
<td>BIT</td>
<td>BIT</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR (n)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
### Table 67. Sybase data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Sybase Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECIMAL</td>
<td>DECIMAL[p, s]</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>IMAGE</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MONEY</td>
<td>DECIMAL[p, s]</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>NTEXT</td>
<td>LONGVARCHAR has a maximum length of 2^31-1 or 2,147,483,647 characters</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC[p, s]</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>DECIMAL[p, s]</td>
</tr>
<tr>
<td>SYSNAME</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>TEXT</td>
<td>LONGVARCHAR has a maximum length of 2^31-1 or 2,147,483,647 characters</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>VARBINARY(8)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARBINARY(n)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>
The following table maps Teradata data types to Composite JDBC data types.

### Table 68. Teradata data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Teradata Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>BYTE</td>
<td>BINARY(n)</td>
</tr>
<tr>
<td>BYTEINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DOUBLE_PRECISION</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>GRAPHIC</td>
<td>CHAR</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTERVAL_DAY</td>
<td>INTERVAL DAY</td>
</tr>
<tr>
<td>INTERVAL_DAY_TO_HOUR</td>
<td>INTERVAL DAY TO HOUR</td>
</tr>
<tr>
<td>INTERVAL_DAY_TO_MINUTE</td>
<td>INTERVAL DAY TO MINUTE</td>
</tr>
<tr>
<td>INTERVAL_DAY_TO_SECOND</td>
<td>INTERVAL DAY TO SECOND</td>
</tr>
<tr>
<td>INTERVAL_HOUR</td>
<td>INTERVAL HOUR</td>
</tr>
<tr>
<td>INTERVAL_HOUR_TO_MINUTE</td>
<td>INTERVAL HOUR TO MINUTE</td>
</tr>
<tr>
<td>INTERVAL_HOUR_TO_SECOND</td>
<td>INTERVAL HOUR TO SECOND</td>
</tr>
<tr>
<td>INTERVAL_MINUTE</td>
<td>INTERVAL MINUTE</td>
</tr>
<tr>
<td>INTERVAL_MINUTE_TO_SECOND</td>
<td>INTERVAL MINUTE TO SECOND</td>
</tr>
</tbody>
</table>

269
MySQL Data Types and Composite JDBC Data Types

Chapter 5  Data Type Mappings

The following table maps MySQL data types to Composite JDBC data types.

In the following table:
- `CHAR(n)` and `VARCHAR(n)` indicate a string column that is allowed to store a maximum of `n` characters.
- `DECIMAL(p, s)` denotes a decimal number with a precision of `p` and a scale of `s`. `p` is the maximum number of digits allowed in the decimal number, including the whole number part and the decimal part (the negative sign and

<table>
<thead>
<tr>
<th>Teradata Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL_MONTH</td>
<td>INTERVAL MONTH</td>
</tr>
<tr>
<td>INTERVAL_SECOND</td>
<td>INTERVAL SECOND</td>
</tr>
<tr>
<td>INTERVAL_YEAR</td>
<td>INTERVAL YEAR</td>
</tr>
<tr>
<td>INTERVAL_YEAR_TO_MONTH</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC{p, s}</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIME WITH_ZONE</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP WITH_ZONE</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR (n)</td>
</tr>
<tr>
<td>VARCHARGRAPHIC</td>
<td>VARCHAR (n)</td>
</tr>
</tbody>
</table>

MySQL Data Types and Composite JDBC Data Types

The following table maps MySQL data types to Composite JDBC data types.

In the following table:
- `CHAR(n)` and `VARCHAR(n)` indicate a string column that is allowed to store a maximum of `n` characters.
- `DECIMAL(p, s)` denotes a decimal number with a precision of `p` and a scale of `s`. `p` is the maximum number of digits allowed in the decimal number, including the whole number part and the decimal part (the negative sign and
the decimal point are not included in \( p \). \( s \) is the maximum number of digits to the right of the decimal point. For example, `DECIMAL(3, 1)` allows a number to range from \(-99.9\) to \(99.9\).

- For all other types, the number in parentheses represents the **display width**, which is the maximum number of digits allowed for a column. For example, `BIGINT(2)` unsigned means that the column is a big integer column whose minimum value is 0 and maximum value is 99.

<table>
<thead>
<tr>
<th>MySQL Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BIT</td>
<td>BIT</td>
</tr>
</tbody>
</table>
| BITINT unsigned | BIGINT when display width is 18 or less.
|                 | NUMERIC(20, 0) when display width is 19 or more. |
| BLOB            | VARBINARY(65535)         |
| BOOL            | BOOLEAN                  |
| CHAR            | CHAR(n)                  |
| CHAR binary     | CHAR(n)                  |
| DATE            | DATE                     |
| DEC             | DECIMAL(p, s)            |
| DEC unsigned    | DECIMAL(p, s)            |
| DECIMAL         | DECIMAL(p, s)            |
| DECIMAL unsigned| DECIMAL(p, s)            |
| DOUBLE          | DOUBLE                   |
| DOUBLE PRECISION| DOUBLE                  |
| DOUBLE PRECISION unsigned | DOUBLE         |
### Table 69. MySQL data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>MySQL Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE unsigned</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>ENUM</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>FIXED</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>FIXED unsigned</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>FLOAT unsigned</td>
<td>FLOAT</td>
</tr>
<tr>
<td>INT or INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT unsigned or INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LONGBLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>LONGTEXT</td>
<td>CLOB</td>
</tr>
<tr>
<td>MEDIUMBLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MEDIUMINT unsigned</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MEDIUMTEXT</td>
<td>CLOB</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>NUMERIC unsigned</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>REAL</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>REAL unsigned</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>SET</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>SMALLINT unsigned</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TEXT</td>
<td>VARCHAR(65535)</td>
</tr>
</tbody>
</table>
Table 69. MySQL data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>MySQL Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TINYBLOB</td>
<td>VARBINARY(255)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>TINYINT unsigned</td>
<td>TINYINT</td>
</tr>
<tr>
<td>TINYTEXT</td>
<td>VARCHAR(255)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>VARCHAR binary</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>YEAR</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>
The following table maps Netezza data types to Composite JDBC data types.

### Table 70. Netezza data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>Netezza Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>BOOL</td>
<td>BIT</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BIT</td>
</tr>
<tr>
<td>BYTETRINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DOUBBLE PRECISION</td>
<td>DOUBBLE</td>
</tr>
<tr>
<td>FLOT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>INT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT1</td>
<td>TINYINT</td>
</tr>
<tr>
<td>INT2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INT4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INT8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>NCHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
</tbody>
</table>
LDAP Data Types and Composite JDBC Data Types

The following table maps the LDAP data type and Composite JDBC data type.

<table>
<thead>
<tr>
<th>LDAP Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMETZ</td>
<td>TIMETZ</td>
</tr>
<tr>
<td>TIME_WITH_TIME_ZONE</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR (n)</td>
</tr>
</tbody>
</table>

CSV Flat File Data Types and Composite JDBC Data Types

The following table maps the CSV flat file data type and Composite JDBC data type.

<table>
<thead>
<tr>
<th>CSV Flat File Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
### MS Access Data Types and Composite JDBC Data Types

The following table maps MS Access data types to Composite JDBC data types.

<table>
<thead>
<tr>
<th>MS Access Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>BIT</td>
</tr>
<tr>
<td>BYTE</td>
<td>TINYINT</td>
</tr>
<tr>
<td>COUNTER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>CURRENCY</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LONGBINARY</td>
<td>BLOB</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(n)</td>
</tr>
</tbody>
</table>
The following table maps MS Excel data types to Composite JDBC data types.

Note that the NUMBER data types returned from Sun’s JDBC ODBC driver do not accurately reflect the real precision and scale if you have formatted the cells in Excel with the following categories: NUMBER, PERCENTAGE, SCIENTIFIC, and FRACTION.

Table 74. MS Excel data types and Composite JDBC data types mapped

<table>
<thead>
<tr>
<th>MS Access Data Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENCY</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>NUMBER</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>VARCHAR(32676)</td>
</tr>
</tbody>
</table>
Chapter 6

JAVA APIs for Custom Procedures

Procedures are used to generate or act on data, much like a SELECT or an UPDATE statement.

This chapter provides Composite’s extended Java APIs that support custom procedures in the system.

- For updated custom Java APIs, see:
  `<Composite installation directory>/apps/extension/docs/index.html`

Examples are available at the end of the appendix on page 303.

All interfaces for custom Java procedures are available in the package:

```
com.compositesw.extension
```

**com.compositesw.extension**

The extension package provides a mechanism for you to write custom procedures.

<table>
<thead>
<tr>
<th>Interface Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomCursor</td>
</tr>
<tr>
<td>Defines a cursor type.</td>
</tr>
<tr>
<td>CustomProcedure</td>
</tr>
<tr>
<td>Defines a custom procedure.</td>
</tr>
<tr>
<td>ExecutionEnvironment</td>
</tr>
<tr>
<td>Used by a procedure to interact with the Composite Server.</td>
</tr>
<tr>
<td>ProcedureConstants</td>
</tr>
<tr>
<td>Constants that are used in the interfaces of the com.compositesw.extension package.</td>
</tr>
</tbody>
</table>
public interface CustomCursor

This interface returns a cursor type. All custom cursors must implement this interface.

A custom procedure with just one output cursor may implement both the 
CustomProcedure and the CustomCursor interfaces to avoid actually needing
another class. A custom procedure with more than one output cursor should use inner classes or separate classes.

### Class Summary

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExecutionEnvironment</td>
<td>Lets a procedure interact with the Composite Server.</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close()</td>
<td>Frees the resources.</td>
</tr>
<tr>
<td>ParameterInfo[]</td>
<td>getColumnInfo()</td>
</tr>
<tr>
<td>Object[]</td>
<td>next()</td>
</tr>
</tbody>
</table>

### Method Detail

#### close

```java
public void close()
    throws CustomProcedureException
```

This method is called when resources should be freed. Calling this method multiple times has no further effect and no exception is thrown.
getColumnInfo

```java
public ParameterInfo[] getColumnInfo()
    throws CustomProcedureException, SQLException
```

This method is called to get the meta-data for the custom cursor. A NULL value may be returned to indicate that the caller should retrieve the meta-data information by calling ProcedureReference.getParameterInfo.

**Returns**
The metadata for the cursor.

**Throws**
- `CustomProcedureException`, if the cursor has been closed or if there is an error fetching the meta-data.
- `SQLException` if there is an error fetching the meta-data.

next

```java
public Object[] next()
    throws CustomProcedureException, SQLException
```

This method is called when more data is needed.

**Returns**
The next row, or NULL when done.

**Throws**
- `CustomProcedureException`, if the cursor has been closed or if there is an error fetching the data.
- `SQLException`, if there is an error fetching the data.
CustomProcedure

```java
public interface CustomProcedure
    extends ProcedureReference
```

This interface defines a custom procedure. Any class implementing this interface should define an empty constructor so that the procedure can be properly instantiated. The `CustomProcedure` interface extends the `ProcedureReference` interface.

All methods on the `CustomProcedure` except for the constructor may throw a `CustomProcedureException` if they encounter an error condition. Any exception thrown from these methods (including runtime exceptions) will result in an error on the current action being passed up as a system error.

**Serialization**: The custom procedure class may implement the `java.lang.Serializable` interface in order to carry compensation state across a server restart. Variables that do not need to be restored after a restart should be marked as `transient`.

**Lifecycle**: The lifecycle of a custom procedure object is defined as follows:

- **Introspection Time** - Constructor is used to make an object, then introspection methods are used to read, then the object is discarded.
- **Runtime Setup** - Constructor is used to make a new object, and `initialize` is called.
- **Runtime Execution** - `invoke` is called first, then output parameter values are retrieved and read from, then output values are retrieved. Note that it is legal to do setup and then not invoke at all.
- **Runtime Closing** - If the object was invoked, then either `commit` or `rollback` will be called. The `close` method is always called last even if not invoked.

**Threading**: The `close` method may be called concurrently with any other call such as `invoke` or `getOutputValues`. In this case, any pending methods should immediately throw a `CustomProcedureException`. 
Chapter 6 JAVA APIs for Custom Procedures

commit

```java
public void commit()
throws CustomProcedureException, SQLException
```

This method commits an open transaction.

**Throws**

An exception if invoked for the parent transaction.

getDescription

```java
public String getDescription()
```

This method is called during data source introspection, and gets the description of the procedure. This method should not return NULL.

**Returns**

Description of the procedure.

---

283
**getName**

```java
public String getName()
```

This method is called during data source introspection, and gets the short name of the procedure. The short name may be overridden during data source configuration. This method should not return `NULL`.

**Returns**

The short name of the procedure.

---

**initialize**

```java
public void initialize(ExecutionEnvironment qenv)
```

This method is called once immediately after constructing the class, and initializes the query execution environment (`ExecutionEnvironment`). The `ExecutionEnvironment` contains methods that are executed to interact with the server.

**Parameters**

- `qenv` - Query execution environment

---

**rollback**

```java
public void rollback()
```

This method rolls back an open transaction.

**Throws**

- A `CustomProcedureException` if invoked for the parent transaction.
Custom Procedure Configuration

All the source and class files for custom procedures are stored in the Composite metadata repository.

A CustomProcedure is defined with the following columns:
- type - Java binary - JAR file containing Java classes
- config - Configuration file specifying the procedures, functions, and hooks contained in the JAR file
- source - Optional, additional JAR file containing the source code

CustomProcedureException

public class CustomProcedureException
extends Exception

This exception is thrown by the methods of the extended APIs in the package com.compositesw.extension. For a summary of the extended APIs, see “Interface Summary” on page 278.

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CustomProcedureException()</td>
</tr>
<tr>
<td></td>
<td>CustomProcedureException(String message)</td>
</tr>
<tr>
<td></td>
<td>CustomProcedureException(String message, Throwable cause)</td>
</tr>
<tr>
<td></td>
<td>CustomProcedureException(Throwerable cause)</td>
</tr>
</tbody>
</table>

Constructor Detail
**CustomProcedureException**

public CustomProcedureException()

This is an empty constructor.

---

**CustomProcedureException**

public CustomProcedureException(String message)

This exception is thrown with a description of the error.

**Parameters**

message - Description of the error.

---

**CustomProcedureException**

CustomProcedureException(String message, Throwable cause)

This exception is thrown with the descriptions of the error and the error’s cause.

**Parameters**

message - Description of the error.
cause - Description of the underlying exception.

---

**CustomProcedureException**

CustomProcedureException(Throwable cause)

This exception is thrown with a description of the error’s cause.

**Parameters**

cause - Explanation of what caused the error.
ExecutionEnvironment

public interface ExecutionEnvironment

Provides an interface between a custom procedure and the Composite Server.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
</tr>
<tr>
<td>ExecutionEnvironment</td>
</tr>
<tr>
<td>java.sql.ResultSet</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>String</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>ProcedureReference</td>
</tr>
<tr>
<td>ProcedureReference</td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>

Method Detail

commit

public void commit()

throws CustomProcedureException, SQLException

This method commits an open transaction.

Throws

This method throws CustomProcedureException if invoked for the parent transaction.
**createTransaction**

```java
public ExecutionEnvironment createTransaction(int flags)
```

This method starts an independent transaction. A custom procedure can have multiple independent transactions open at the same time using this method.

**Parameters**
- `flags` - used to pass in transaction options for compensate mode, recovery mode, and recovery level.
  - Legal flag values are:
    - COMPENSATE* | NO_COMPENSATE
    - ROLLBACK* | BEST_EFFORT
    - IGNORE_INTERRUPT* | LOG_INTERRUPT | FAIL_INTERRUPT
  - (The asterisk indicates the default value if no flags are specified.)

**executeQuery**

```java
public java.sql.ResultSet executeQuery (String sql, Object[] args)
```

This method is used to execute a `SELECT` statement from inside the stored procedure. It should not return `NULL`.

**Parameters**
- `sql` - SQL statement
- `args` - Arguments for the query. Can be `NULL` if there are no arguments.
  - The `arg` objects should comply with the Java to SQL typing conventions as defined in the section "Types" on page 301. Input cursors are accepted as both `CustomCursor` and `java.sql.ResultSet`.
**executeUpdate**

```java
public int executeUpdate (String sql)
    throws CustomProcedureException
```

Used to execute an `INSERT`, `UPDATE`, or `DELETE` statement from inside the stored procedure call.

**Parameters**
- `sql` - SQL statement

**Throws**
- `CustomProcedureException` if there is a problem executing the `sql`.

**Returns**
- Number of rows affected.
- `-1` if affected number of rows is unknown

**getProperty**

```java
public String getProperty(String name)
```

This method is used to get environmental properties.

**Parameters**
- `name` - property

**Returns**
- `null` if the property is not defined.

**log**

```java
public void log(int level, String st)
```
This method sends an entry to the system log.

**Parameters**
- `st` - log entry
  - Level: `ERROR`, `INFO`, or `DEBUG`

### lookupNextHook

```java
public ProcedureReference lookupNextHook() throws CustomProcedureException
```

This method is used by hook procedures to invoke the next hook in the list. It should not return `NULL`.

### lookupProcedure

```java
public ProcedureReference lookupProcedure(String procedureName) throws CustomProcedureException
```

This method is used to look up a procedure reference from the query. The `close` method must be called on the returned procedure when it is no longer needed. This method will not return `NULL`.

**Parameters**
- `procedureName` - name of the procedure

**Throws**
- `CustomProcedureException` if the procedure is not found.

### rollback

```java
public void rollback() throws CustomProcedureException, SQLException
```

This method rolls back an open transaction.

**Throws**
This method throws CustomProcedureException if invoked for the parent transaction.

ParameterInfo

```java
public class ParameterInfo {
    
    This class is used to get the description of the procedure's input and output parameters.

    Constructor Summary

    ParameterInfo (String name, int type)
        Creates a new ParameterInfo with the specified parameter values.

    ParameterInfo (String name, int type, int direction)

    ParameterInfo (String name, int type, int direction, ParameterInfo[] columns)

    ParameterInfo (String name, int type, int direction, String xmlSchema)

    Method Summary

    ParameterInfo[] getColumn()

    int getDirection ()

    String getName()  

    int getType()

    String getXmlSchema()  

}```
ParameterInfo

public ParameterInfo (String name,
               int type)

Creates a new ParameterInfo with the specified parameter values.

Parameters

name - Name of the column or parameter

type - Types are from java.sql.Types, with XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR

ParameterInfo

public ParameterInfo (String name,
               int type,
               int direction)

Creates a new ParameterInfo with the specified parameter values.

Parameters

name - Name of the column or parameter

type - Types are from java.sql.Types, with XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR

direction - The direction may be DIRECTION_IN, DIRECTION_INOUT, or DIRECTION_OUT. This value is not relevant for column definitions.
**ParameterInfo**

```java
public ParameterInfo (String name,
                     int type,
                     int direction,
                     (ParameterInfo[]) columns)
```

Creates a new ParameterInfo with the specified parameter values.

**Parameters**

- **name** - Name of the column or parameter
- **type** - Types are from `java.sql.Types`, with `XML_STRING`, `TYPED_CURSOR`, and `GENERIC_CURSOR`
- **direction** - The direction may be `DIRECTION_IN`, `DIRECTION_INOUT`, or `DIRECTION_OUT`. This value is not relevant for column definitions.
- **columns** - Non-null, if the type is `TYPED_CURSOR`

---

```java
public ParameterInfo (String name,
                     int type,
                     int direction,
                     String xmlSchema)
```

Creates a new ParameterInfo with the specified parameter values.

**Parameters**

- **name** - Name of the column or parameter
- **type** - Types are from `java.sql.Types`, with `XML_STRING`, `TYPED_CURSOR`, and `GENERIC_CURSOR`
- **direction** - The direction may be `DIRECTION_IN`, `DIRECTION_INOUT`, or `DIRECTION_OUT`. This value is not relevant for column definitions.
- **xmlSchema** - Non-null, if the type is `XML_STRING`
getColumns

public ParameterInfo[] getColumns()
This method gets the columns if the type is TYPED_CURSOR

Returns
The columns if the column data type is TYPED_CURSOR.

getDirection

public int getDirection()
This method gets the direction of the parameter.

Returns
The direction of the parameter.
The direction may be DIRECTION_IN, DIRECTION_INOUT, or DIRECTION_OUT.

getName

public String getName()
This method gets the name of the column or parameter.

Returns
The name of the column or parameter.
**getType**

public int getType()

This method gets the type of the column or parameter.

**Returns**

The type of the column or parameter. The types are from java.sql.Types, with the addition of XML_STRING, TYPED_CURSOR, and GENERIC_CURSOR.

**getXmlSchema**

public String getXmlSchema()

This method gets the schema, if the type is XML_STRING.

**Returns**

The schema, if the type is XML_STRING.

**ProcedureConstants**

public interface ProcedureConstants

This interface implements the constants that are used in the interfaces of the com.compositesw.extension package. For a summary of the extended APIs, see “Interface Summary” on page 278.

<table>
<thead>
<tr>
<th>Field Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>int DIRECTION_IN</td>
</tr>
<tr>
<td>int DIRECTION_INOUT</td>
</tr>
<tr>
<td>int DIRECTION_NONE</td>
</tr>
<tr>
<td>int DIRECTION_OUT</td>
</tr>
<tr>
<td>int GENERIC_CURSOR</td>
</tr>
<tr>
<td>int HOOK_TYPE_SQL</td>
</tr>
</tbody>
</table>
### Field Summary

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>HOOK_TYPE_PROCEDURE</td>
</tr>
<tr>
<td>int</td>
<td>LOG_DEBUG</td>
</tr>
<tr>
<td>int</td>
<td>LOG_ERROR</td>
</tr>
<tr>
<td>int</td>
<td>LOG_INFO</td>
</tr>
<tr>
<td>int</td>
<td>TXN_BEST_EFFORT</td>
</tr>
<tr>
<td>int</td>
<td>TXN_COMPENSATE</td>
</tr>
<tr>
<td>int</td>
<td>TXN_NO_COMPENSATE</td>
</tr>
<tr>
<td>int</td>
<td>TXN.Rollback</td>
</tr>
<tr>
<td>int</td>
<td>TXN_IGNORE_INTERRUPT</td>
</tr>
<tr>
<td>int</td>
<td>TXN_LOG_INTERRUPT</td>
</tr>
<tr>
<td>int</td>
<td>TXN_FAIL_INTERRUPT</td>
</tr>
<tr>
<td>int</td>
<td>TYPED_CURSOR</td>
</tr>
<tr>
<td>int</td>
<td>XML_STRING</td>
</tr>
</tbody>
</table>

### Field Detail

**DIRECTION_IN**

```java
public static final int DIRECTION_IN
```

IN parameter direction constant.

**DIRECTION_INOUT**

```java
public static final int DIRECTION_IN
```


INOUT parameter direction constant.

**DIRECTION_NONE**

```java
public static final int DIRECTION_NONE = 0
```

NONE parameter direction constant.

This constant is used for `ParameterInfo` objects that represent columns in a cursor. See `ProcedureReference.getParameterInfo`.

**DIRECTION_OUT**

```java
public static final int DIRECTION_OUT
```

OUT parameter direction constant.

**GENERIC_CURSOR**

```java
public static final int GENERIC_CURSOR = 5520;
```

Type constant for a cursor whose schema is resolved at runtime.

**LOG_DEBUG**

```java
public static final int LOG_DEBUG
```

Debug logging level (3).

**LOG_ERROR**

```java
public static final int LOG_ERROR
```

Debug logging level (1).

**LOG_INFO**

```java
public static final int LOG_INFO
```
Debug logging level (2).

**TYPED_CURSOR**
public static final int TYPED_CURSOR = 5521;
Type constant for a cursor with accompanying meta-data.

**TXN_BEST_EFFORT**
public static final int TXN_BEST_EFFORT
Best effort transaction flag.

**TXN_FAIL_INTERRUPT**
public static final int TXN_FAIL_INTERRUPT
Fail interrupt transaction flag.

**TXN_IGNORE_INTERRUPT**
public static final int TXN_IGNORE_INTERRUPT
Ignore interrupt transaction flag.

**TXN_LOG_INTERRUPT**
public static final int TXN_LOG_INTERRUPT
Log interrupt transaction flag.

**TXN_NO_COMPENSATE**
public static final int TXN_NO_COMPENSATE
No compensation transaction flag.
**TXN_ROLLBACK**

public static final int TXN_ROLLBACK

Rollback transaction flag.

**XML_STRING**

public static final int XML_STRING = 5500;

Type constant for hierarchical XML data.

### ProcedureReference

The `ProcedureReference` interface provides a way to invoke a procedure and fetch its output values. It also provides meta-data information for the procedure parameters. `ProcedureReference` is a parent interface for the `CustomProcedure` interface. It is also used as the return type when looking up a procedure from the query engine.

The type of each Java object must be the default Java object type corresponding to the input or output parameter’s SQL type, following the mapping for built-in types specified in the JDBC specification (as per the `getObject` method on `java.sql.ResultSet`). See Table 75, “Java Object Types Mapped to Composite JDBC Data Types” on page 301.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>Object[]</td>
</tr>
<tr>
<td>ParameterInfo[]</td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>
**close**

public void close()

This method is called when the procedure reference is no longer needed. This method may be called concurrently with any other call such as `invoke` or `getOutputValues`. When called concurrently with another call such as `invoke` or `getOutputValues`, this method should cause a `CustomProcedureException`.

The implementation of this method should close all open cursors and all independent transactions that this method has created.

**getNumAffectedRows**

public int getNumAffectedRows() throws CustomProcedureException, SQLException

This method is called to retrieve the number of rows that were inserted, updated, or deleted during the execution of a procedure.

**Returns**

A return value of -1 indicates that the number of affected rows is unknown.

**Throws**

Throws `CustomProcedureException` or `SQLException` if there is an error when getting the number of affected rows.

**getOutputValues**

public Object[] getOutputValues() throws CustomProcedureException, SQLException
This method is called to retrieve a procedure’s output values. This method should not return NULL. The returned objects should comply with the Java to SQL typing conventions as defined in the section “Types” on page 301.

**Returns**

This method can return output cursors as either `CustomCursor` or `java.sql.ResultSet`.

**Throws**

This method can throw `CustomProcedureException` or `SQLException` if there is an error when getting the output values.

**Types**

The `getOutputValues` method (page 300) of the `ProcedureReference` interface retrieves the output values in a procedure. The returned objects should comply with the Java to SQL typing conventions as defined here.

The type of each Java object must be the default Java object type corresponding to the input or output parameter’s Composite JDBC data type, following the mapping for built-in types specified in the JDBC specification (as per the `getObject` method on `java.sql.ResultSet`).

The following table maps the Java object types to Composite JDBC data types:

<table>
<thead>
<tr>
<th>Java Object Type</th>
<th>Composite JDBC Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.String</code></td>
<td>CHAR, VARCHAR, or LONGVARCHAR</td>
</tr>
<tr>
<td><code>java.math.BigDecimal</code></td>
<td>NUMERIC or DECIMAL</td>
</tr>
<tr>
<td><code>java.lang.Boolean</code></td>
<td>BIT or BOOLEAN</td>
</tr>
<tr>
<td><code>java.lang.Integer</code></td>
<td>INTEGER, SMALLINT, or TINYINT</td>
</tr>
<tr>
<td><code>java.lang.Long</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>java.lang.Float</code></td>
<td>REAL or FLOAT</td>
</tr>
<tr>
<td><code>java.lang.Double</code></td>
<td>DOUBLE</td>
</tr>
<tr>
<td><code>byte[]</code></td>
<td>BINARY, VARBINARY, or LONGVARBINARY</td>
</tr>
</tbody>
</table>
Special Types and Values

- If the input or output parameter type is XML_STRING, then the Java object type should be java.lang.String.
- If the parameter type is TYPED_CURSOR or GENERIC_CURSOR, then for input parameters the Java object type will always be java.sql.ResultSet, and for output parameters the Java object type is allowed to be either CustomCursor or java.sql.ResultSet.
- If the value is an SQL NULL, the procedure returns a Java NULL.

Hierarchical Data

This interface is primarily designed around tabular data. A stored procedure that has hierarchical input or output should accept or return one or more scalar parameters that contain XML string data. For methods that use java.sql.Types, the constant XML_STRING should be used for hierarchical XML data.

Cursors

The types TYPED_CURSOR and GENERIC_CURSOR are used to pass in and out cursor values. A typed cursor has a schema. A generic cursor’s schema is resolved at runtime. Procedures with generic cursor outputs cannot be used in SQL. They can only be used in composition or from JDBC/ODBC.

getParameterInfo

public ParameterInfo[] getParameterInfo()
This method is called during introspection to get the description of the procedure’s input and output parameters. This method should not return NULL.

**Returns**

Returns the description of the procedure’s input and output parameters.

### invoke

```java
public void invoke(Object[] inputValues)
    throws CustomProcedureException, SQLException
```

This method is called to invoke a procedure. It is called only once per procedure instance.

**Parameters**

- `inputValues`: values for the input parameters. Must not be NULL.

**Throws**

- Throws `CustomProcedureException` or `SQLException` if there is an error during invocation.

### Examples

This section contains several examples to illustrate the behavior of a custom procedure.

- **“Example 1 - Simple Query” on page 305**
  
  Here, the custom procedure participates in the parent transaction, and invokes a query using the execution environment.

- **“Example 2 - Simple Update” on page 308**
  
  Here, the custom procedure participates in the parent transaction, and performs an update using the execution environment.

- **“Example 3 - External Update” on page 312**
Examples

Chapter 6  JAVA APIs for Custom Procedures

Here, the custom procedure uses an independent transaction with a transactional data source in the server. Compensating logic is defined for the independent transaction.

“Example 4 - Non-Transactional” on page 317
Here, the custom procedure updates the contents of a file on disk where the file is non-transactional. The actual work is deferred until the commit method is called. Compensating logic is provided.

“Example 5 - Expression Evaluator” on page 322
Here, the custom procedure evaluates simple expressions.

“Example 6 - Output Cursor” on page 327
Here, the custom procedure invokes another procedure, and retrieves output values.

“Example 7 - Simple Procedure Invoke” on page 332
Here, the custom procedure invokes another procedure.
Example 1 - Simple Query
The custom procedure participates in the parent transaction, and invokes a query using the execution environment.

```java
/**
 * Custom Procedure Examples
 */
package proc;
import com.compositesw.extension.*;
import java.sql.*;

/**
 * This custom procedure executes a simple query statement
 */
public class SimpleQuery
    implements CustomProcedure
{
    private ExecutionEnvironment qenv;
    private ResultSet resultSet;
    public SimpleQuery() {}
    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }
    public ParameterInfo[] getParameterInfo() {
        return new ParameterInfo[] {
            new ParameterInfo("id", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("result", TYPES_CURSOR, DIRECTION_OUT),
        };
    }
```
new ParameterInfo("Id", Types.INTEGER, DIRECTION_NONE),
new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_NONE),
new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_NONE),
new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_NONE),
new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_NONE),
};

/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */
public void invoke(Object[] inputValues)
throws CustomProcedureException, SQLException
{
    resultSet = qenv.executeQuery(
        "SELECT " +
        "CustomerID AS Id, " +
        "ContactFirstName AS FirstName, " +
        "ContactLastName AS LastName, " +
        "CompanyName AS CompanyName, " +
        "PhoneNumber AS PhoneNumber FROM " +
        "/shared/tutorial/sources/ds_orders/customers WHERE CustomerID=" +
        inputValues[0],
    null);
}
/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected row is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows()
{
    return 0;
}
/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues() {
    return new Object[] { resultSet };
}
/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */
public void close() throws SQLException {
    if (resultSet != null) {
        resultSet.close();
    }
}

// Introspection methods
/**
 * Called during introspection to get the short name of the stored
 * procedure. This name may be overridden during configuration.
 * Should not return null.
 */
public String getName() {
    return "SimpleQuery";
}
/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */
public String getDescription() {
    return "This procedure performs a simple query operation";
}

// Transaction methods
/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */
public boolean canCommit() {
    return false;
Examples

Chapter 6  JAVA APIs for Custom Procedures

```java
/**
 * Commit any open transactions.
 */
public void commit() {
}

/**
 * Rollback any open transactions.
 */
public void rollback() {
}

/**
 * Returns true if the transaction can be compensated.
 */
public boolean canCompensate() {
    return false;
}

/**
 * Compensate any committed transactions (if supported).
 */
public void compensate(ExecutionEnvironment qenv) {
}
```

Example 2 - Simple Update

The custom procedure participates in the parent transaction, and performs an update using the execution environment.

```java
/**
 * This custom procedure executes a simple update statement
 */
public class SimpleUpdate
```
implements CustomProcedure
{
    private ExecutionEnvironment qenv;
    private int numRowsUpdated = -1;
    public SimpleUpdate() {}

    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */
    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }

    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */
    public ParameterInfo[] getParameterInfo() {
        return new ParameterInfo[] {
            new ParameterInfo("Id", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_IN),
        };
    }

    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     * SQLException if there is an error during invoke.
     */
    public void invoke(Object[] inputValues)
        throws CustomProcedureException, SQLException
    {
        // Update in the first data source using an SQL statement
        numRowsUpdated = qenv.executeUpdate("UPDATE /shared/tutorial/sources/ds_orders/customers" +
            " SET ContactFirstName='" + inputValues[1] +
            "', ContactLastName='" + inputValues[2] +
            ", CompanyName='" + inputValues[3] +
            "', PhoneNumber='" + inputValues[4] +
            
            WHERE CustomerID='" + inputValues[0],
            null);
    }
Examples

Chapter 6  JAVA APIs for Custom Procedures

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows() {
    return numRowsUpdated;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues() {
    return new Object[] { }
}

/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as 'invoke' or 'getOutputValues'. In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */
public void close() { }

/**
 * Called during introspection to get the short name of the stored
 * procedure. This name may be overridden during configuration.
 * Should not return null.
 */
public String getName() {
    return "SimpleUpdate";
}
public String getDescription() {
    return "This procedure performs a simple update operation";
}

public boolean canCommit() {
    return false;
}

public void commit() {
}

public void rollback() {
}

public boolean canCompensate() {
    return false;
}

public void compensate(ExecutionEnvironment qenv) {
}
Example 3 - External Update

The custom procedure uses an independent transaction with a transactional data source in the server. Compensating logic is defined for the independent transaction.

```java
/**
 * Custom Procedure Examples
 */
package proc;
import com.compositesw.extension.*;
import java.sql.*;
/**
 * External update example with compensation
 */
public class ExternalUpdate
    implements CustomProcedure, java.io.Serializable
{
    private static final String ORDERS_URL =
        "jdbc:mysql://localhost:3306/Orders";
    private transient ExecutionEnvironment qenv;
    private transient Connection conn;
    private transient int numRowsUpdated;
    private boolean isUpdate;
    private int id;
    private String firstName;
    private String lastName;
    private String companyName;
    private String phoneNumber;
    public ExternalUpdate() {} 
    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */
    public void initialize(ExecutionEnvironment qenv)
        throws SQLException
    {
        this.qenv = qenv;
        conn = DriverManager.getConnection(ORDERS_URL, "tutorial", "tutorial");
    }
}
```
conn.setAutoCommit(false);
}
/**
* Called during introspection to get the description of the input
* and output parameters. Should not return null.
*/
public ParameterInfo[][] getParameterInfo() {
    return new ParameterInfo[][] {
        new ParameterInfo[] {"Id", Types.INTEGER, DIRECTION_IN},
        new ParameterInfo[] {"FirstName", Types.VARCHAR, DIRECTION_IN},
        new ParameterInfo[] {"LastName", Types.VARCHAR, DIRECTION_IN},
        new ParameterInfo[] {"CompanyName", Types.VARCHAR, DIRECTION_IN},
        new ParameterInfo[] {"PhoneNumber", Types.VARCHAR, DIRECTION_IN},
    };
}
/**
* Called to invoke the stored procedure. Will only be called a
* single time per instance. Can throw CustomProcedureException or
* SQLException if there is an error during invoke.
*/
public void invoke(Object[] inputValues)
    throws CustomProcedureException, SQLException
{
    Statement stmt = conn.createStatement();
    //
    // Save away the current values to be used for compensation
    //
    ResultSet rs = stmt.executeQuery("SELECT ContactFirstName, ContactLastName, CompanyName, PhoneNumber " +
        "FROM customers WHERE CustomerID=" + inputValues[0]);
    if (rs.next()) {
        isUpdate = true;
        id = ((Integer)inputValues[0]).intValue();
        firstName = rs.getString(1);
        lastName = rs.getString(2);
        companyName = rs.getString(3);
        phoneNumber = rs.getString(4);
    }
    rs.close();
    //
    // Perform the insert or update
    //
if (isUpdate) {
  numRowsUpdated = stmt.executeUpdate(
    "UPDATE customers" +
    " SET ContactFirstName='" + inputValues[1] +
    "', ContactLastName='" + inputValues[2] +
    ", CompanyName='" + inputValues[3] +
    ", PhoneNumber='" + inputValues[4] +
    " WHERE CustomerID='" + inputValues[0] +
  );
} else {
  numRowsUpdated = stmt.executeUpdate(
    "INSERT into customers (CustomerID, ContactFirstName, " +
    "ContactLastName, CompanyName, PhoneNumber) VALUES (" +
    inputValues[0] + ", ", " + inputValues[1] + ", ", " +
    inputValues[4] + ");
  }
}
}

if (isUpdate) {
  numRowsUpdated = stmt.executeUpdate(
    "UPDATE customers" +
    " SET ContactFirstName='" + inputValues[1] +
    "', ContactLastName='" + inputValues[2] +
    ", CompanyName='" + inputValues[3] +
    ", PhoneNumber='" + inputValues[4] +
    " WHERE CustomerID='" + inputValues[0] +
  );
} else {
  numRowsUpdated = stmt.executeUpdate(
    "INSERT into customers (CustomerID, ContactFirstName, " +
    "ContactLastName, CompanyName, PhoneNumber) VALUES (" +
    inputValues[0] + ", ", " + inputValues[1] + ", ", " +
    inputValues[4] + ");
  }
}

public int getNumAffectedRows() {
  return numRowsUpdated;
}

public Object[] getOutputValues() {
  return new Object[] { }
}

/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
public void close()
        throws SQLException
        {
        }

    // Introspection methods
    
    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name may be overridden during configuration.
     * Should not return null.
     */
    public String getName() {
        return "ExternalUpdate";
    }

    /**
     * Called during introspection to get the description of the stored
     * procedure. Should not return null.
     */
    public String getDescription() {
        return "This procedure performs an update to an external transactional database source using JDBC.";
    }

    // Transaction methods
    
    /**
     * Returns true if the custom procedure uses transactions. If this
     * method returns false then commit and rollback will not be called.
     */
    public boolean canCommit() {
        return true;
    }

    /**
     * Commit any open transactions
     */
    
    // cleanup. Close may be called concurrently with any other call
    * such as "invoke" or "getOutputValues". In this case, any pending
    * methods should immediately throw a CustomProcedureException.
    */
public void commit() throws SQLException {
    conn.commit();
    conn.close();
    conn = null;
}

/**
 * Rollback any open transactions.
 */
public void rollback() throws SQLException {
    conn.rollback();
    conn.close();
    conn = null;
}

/**
 * Returns true if the transaction can be compensated.
 */
public boolean canCompensate() {
    return true;
}

/**
 * Compensate any committed transactions (if supported).
 */
public void compensate(ExecutionEnvironment qenv) throws SQLException {
    conn = DriverManager.getConnection(ORDERS_URL);
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    if (isUpdate) {
        numRowsUpdated = stmt.executeUpdate("UPDATE customers
                                         SET ContactFirstName='" + firstName + 
                                         "', ContactLastName='" + lastName + 
                                         "', CompanyName='" + companyName + 
                                         "', PhoneNumber='" + phoneNumber + 
                                         " WHERE CustomerID=" + id;
    }
    else {
        // Remaining code
Example 4 - Non-Transactional

The custom procedure updates the contents of a file on disk where the file is non-transactional. The actual work is deferred until the commit method is called. Compensating logic is provided.

```java
import com.compositesw.extension.*;
import java.sql.*;
import java.io.*;

/**
 * Non-transactional external update example with compensation
 *
 */
public class NonTransactional
    implements CustomProcedure, java.io.Serializable
{
    private transient ExecutionEnvironment qenv;
    private transient File dataFile;
    private transient int numRowsUpdated;
    private transient int newId;
    private transient String newFirstName;
    private transient String newLastName;
    private transient String newCompanyName;
    private transient String newPhoneNumber;
    private int oldId;
    private String oldFirstName;
    private String oldLastName;
    private String oldCompanyName;
    private String oldPhoneNumber;

    stmt.executeUpdate("DELETE from customers WHERE CustomerID=" + id);
    stmt.close();
    conn.commit();
    conn.close();
    conn = null;
}
```
public NonTransactional() {
}
/**
 * This is called once just after constructing the class. The
 * environment contains methods used to interact with the server.
 */
public void initialize(ExecutionEnvironment qenv)
    throws CustomProcedureException {
    this.qenv = qenv;
    dataFile = new File("C:/CustomProcNonTrans.txt");
    try {
        if (!dataFile.canWrite() && !dataFile.createNewFile())
            throw new CustomProcedureException("cannot write file");
    } catch (IOException ex) {
        throw new CustomProcedureException(ex);
    }
    catch (IOException ex) {
        throw new CustomProcedureException(ex);
    }
}
/**
 * Called during introspection to get the description of the input
 * and output parameters. Should not return null.
 */
public ParameterInfo[] getParameterInfo()
    throws CustomProcedureException {
    return new ParameterInfo[] {
        new ParameterInfo("Id", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("FirstName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("LastName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("CompanyName", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("PhoneNumber", Types.VARCHAR, DIRECTION_IN)
    };
}
/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */
public void invoke(Object[] inputValues)
    throws CustomProcedureException {
    //
    // Save new values for later use in 'commit'
    //
    //
newId = ((Integer)inputValues[0]).intValue();
newFirstName = (String)inputValues[1];
newLastName = (String)inputValues[2];
newCompanyName = (String)inputValues[3];
newPhoneNumber = (String)inputValues[3];
}
/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows()
throws CustomProcedureException
{
  return numRowsUpdated;
}
/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues()
throws CustomProcedureException
{
  return new Object[] { };}
/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */
public void close() { }

// Introspection methods
//
Examples

Chapter 6  JAVA APIs for Custom Procedures

/**
 * Called during introspection to get the short name of the stored
 * procedure. This name may be overridden during configuration.
 * Should not return null.
 */
public String getName() {
    return "NonTransactional";
}

/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */
public String getDescription() {
    return "This procedure performs an update to an external " +
            "non-transactional file data source.";
}

// Transaction methods

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */
public boolean canCommit() {
    return true;
}

/**
 * Commit any open transactions.
 */
public void commit()
    throws CustomProcedureException
{
    // Save away the current values to be used for compensation
    try {
        BufferedReader reader = new BufferedReader(new FileReader(dataFile));
        String line = reader.readLine();
        oldId = (line == null || line.length() == 0) ? 0 : Integer.parseInt(line);
    }
    ...
oldFirstName = reader.readLine();
oldLastName = reader.readLine();
oldCompanyName = reader.readLine();
oldPhoneNumber = reader.readLine();
reader.close();
} catch (IOException ex) {
    throw new CustomProcedureException(ex);
}

// Write the new data out to the file
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(dataFile));
    writer.write(Integer.toString(newId)); writer.newLine();
    writer.write(newFirstName); writer.newLine();
    writer.write(newLastName); writer.newLine();
    writer.write(newCompanyName); writer.newLine();
    writer.write(newPhoneNumber); writer.newLine();
    writer.close();
} catch (IOException ex) {
    throw new CustomProcedureException(ex);
}

/**
 * Rollback any open transactions.
 */
public void rollback() {
    // do nothing
}

/**
 * Returns true if the transaction can be compensated.
 */
public boolean canCompensate() {
    return true;
}

/**
 * Compensate any committed transactions (if supported).
 */
public void compensate(ExecutionEnvironment qenv)
    throws CustomProcedureException {
    // code
}
// Restore the old data
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(dataFile));
    writer.write(Integer.toString(oldId)); writer.newLine();
    writer.write(oldFirstName); writer.newLine();
    writer.write(oldLastName); writer.newLine();
    writer.write(oldCompanyName); writer.newLine();
    writer.write(oldPhoneNumber); writer.newLine();
    writer.close();
} catch (IOException ex) {
    throw new CustomProcedureException(ex);
}
}

Example 5 - Expression Evaluator

/**
 * Custom Procedure Examples
 */
package proc;

import com.compositesw.extension.*;
import java.sql.SQLException;
import java.sql.Types;

/**
 * Custom procedure to evaluate simple expressions:
 */
ARGS1 | ARGS2
ARGS1 if it is neither null nor 0, otherwise ARGS2
ARGS1 & ARGS2
ARGS1 if neither argument is null or 0, otherwise 0
ARGS1 < ARGS2
ARGS1 is less than ARGS2
ARGS1 <= ARGS2
ARGS1 is less than or equal to ARGS2
ARGS1 = ARGS2
ARGS1 = ASRGS1

322
Examples

Chapter 6  JAVA APIs for Custom Procedures

* ARG1 is equal to ARG2
* ARG1 := ARG2
* ARG1 is unequal to ARG2
* ARG1 >= ARG2
* ARG1 is greater than or equal to ARG2
* ARG1 > ARG2
* ARG1 is greater than ARG2
* ARG1 + ARG2
* arithmetic sum of ARG1 and ARG2
* ARG1 - ARG2
* arithmetic difference of ARG1 and ARG2
* ARG1 * ARG2
* arithmetic product of ARG1 and ARG2
* ARG1 / ARG2
* arithmetic quotient of ARG1 divided by ARG2
* ARG1 % ARG2
* arithmetic remainder of ARG1 divided by ARG2

*/

public class ExpressionEvaluator

implements CustomProcedure

private ExecutionEnvironment qenv;
private int result;
public ExpressionEvaluator() { }
/**
 * This is called once just after constructing the class. The
 * environment contains methods used to interact with the server.
 */

public void initialize(ExecutionEnvironment qenv)
throws SQLException
{
this.qenv = qenv;
}
/**
 * Called during introspection to get the description of the input
 * and output parameters. Should not return null.
 */
public ParameterInfo[] getParameterInfo() {
    return new ParameterInfo[] {
        new ParameterInfo("arg1", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("operator", Types.VARCHAR, DIRECTION_IN),
        new ParameterInfo("arg2", Types.INTEGER, DIRECTION_IN),
        new ParameterInfo("result", Types.INTEGER, DIRECTION_OUT),
    };
}

/**
 * Called to invoke the stored procedure. Will only be called a single
 * time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */
public void invoke(Object[] inputValues)
    throws CustomProcedureException, SQLException
{
    int arg1 = (inputValues[0] != null ? ((Integer)inputValues[0]).intValue() : 0);
    String op = (String)inputValues[1];
    int arg2 = (inputValues[2] != null ? ((Integer)inputValues[2]).intValue() : 0);
    if (op.equals("|"))
        result = (arg1 != 0) ? arg1 : arg2;
    else if (op.equals("&"))
        result = (arg1 != 0 && arg2 != 0) ? arg1 : 0;
    else if (op.equals(`<`))
        result = (arg1 < arg2) ? 1 : 0;
    else if (op.equals("<="))
        result = (arg1 <= arg2) ? 1 : 0;
    else if (op.equals("="))
        result = (arg1 == arg2) ? 1 : 0;
    else if (op.equals("!="))
        result = (arg1 != arg2) ? 1 : 0;
    else if (op.equals(">="))
        result = (arg1 >= arg2) ? 1 : 0;
    else if (op.equals(">"))
        result = (arg1 > arg2) ? 1 : 0;
    else if (op.equals("+"))
        result = arg1 + arg2;
    else if (op.equals("-"))
        result = arg1 - arg2;
    else if (op.equals("*"))
        result = arg1 * arg2;
    else if (op.equals(`/`))
        result = arg1 / arg2;
    else if (op.equals("%"))
        result = arg1 % arg2;
    else
        result = 0;
}
throw new CustomProcedureException("Unknown operator: " + op);
}
/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows() {
    return 0;
}
/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues() {
    return new Object[] { new Integer(result) };
}
/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */
public void close() throws SQLException
{
    //
    // Introspection methods
    //
    /**
     * Called during introspection to get the short name of the stored
     * procedure. This name may be overridden during configuration.
     * Should not return null.
     */
}
public String getName() {
    return "expr";
}
/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */
public String getDescription() {
    return "Custom procedure to evaluate simple expressions";
}

// Transaction methods

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */
public boolean canCommit() {
    return false;
}
/**
 * Commit any open transactions.
 */
public void commit() throws SQLException {
}
/**
 * Rollback any open transactions.
 */
public void rollback() throws SQLException {
}
/**
 * Returns true if the transaction can be compensated.
 */
public boolean canCompensate() {
    return false;
}
/**
public void compensate(ExecutionEnvironment qenv)
        throws SQLException
    {
    }

Example 6 - Output Cursor

    /**
     * Custom Procedure Examples
     */
    package proc;
    import com.compositesw.extension.*;
    import java.sql.SQLException;
    import java.sql.Timestamp;
    import java.sql.Types;
    public class OutputCursor
        implements CustomProcedure, java.io.Serializable
    {
        private transient ExecutionEnvironment qenv;
        private transient CustomCursor outputCursor;
        private boolean invoked;
        public OutputCursor() { }

        /**
         * This is called once just after constructing the class. The
         * environment contains methods used to interact with the server.
         */
        public void initialize(ExecutionEnvironment qenv)
            throws SQLException
        {
            this.qenv = qenv;
        }

        /**
         * Called during introspection to get the description of the input
         * and output parameters. Should not return null.
         */
public ParameterInfo[] getParameterInfo() {
  return new ParameterInfo[] {
    new ParameterInfo("result", TYPED_CURSOR, DIRECTION_OUT,
      new ParameterInfo("IntColumn", Types.INTEGER, DIRECTION_NONE),
      new ParameterInfo("StringColumn", Types.VARCHAR, DIRECTION_NONE),
      new ParameterInfo("TimestampColumn", Types.TIMESTAMP, DIRECTION_NONE),
    )
  };
}

/**
 * Called to invoke the stored procedure. Will only be called a
 * single time per instance. Can throw CustomProcedureException or
 * SQLException if there is an error during invoke.
 */
public void invoke(Object[] inputValues)
  throws CustomProcedureException, SQLException
{
  invoked = true;
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows() {
  return 0;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues() {
  outputCursor = createCustomCursor();
  return new Object[] { outputCursor };
}
/**
 * Create a custom cursor output.
 */

private static CustomCursor createCustomCursor() {
    return new CustomCursor() {
        private int counter;
        public ParameterInfo[] getColumnInfo() {
            return null;
        }
        public Object[] next() throws CustomProcedureException, SQLException {
            return new Object[] {
                new Integer(counter),
                Integer.toString(counter),
                new Timestamp(counter),
            };
        }
        public void close() throws CustomProcedureException, SQLException {
            if (outputCursor != null)
                outputCursor.close();
        }
    };
}

/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as “invoke” or “getOutputValues”. In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */

public void close() throws CustomProcedureException, SQLException {
    if (outputCursor != null)
        outputCursor.close();
}

329
// Introspection methods
/**
 * Called during introspection to get the short name of the stored
 * procedure. This name may be overridden during configuration.
 * Should not return null.
 */
public String getName() {
    return "OutputCursor";
}
/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */
public String getDescription() {
    return "Custom procedure that returns cursor data";
}

// Transaction methods
/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */
public boolean canCommit() {
    return true;
}
/**
 * Commit any open transactions.
 */
public void commit() throws SQLException {
}
/**
 * Rollback any open transactions.
 */
public void rollback() throws SQLException {
}
public boolean canCompensate() {
    return true;
}

public void compensate(ExecutionEnvironment qenv)
    throws SQLException {
    System.out.println("OutputCursor.compensate(): invoked=" + invoked);
}
Example 7 - Simple Procedure Invoke

```java
/**
 * Custom Procedure Examples
 */
package proc;
import com.compositesw.extension.*;
import java.sql.*;
/**
 * This custom procedure invokes another procedure.
 */
public class SimpleProcInvoke
    implements CustomProcedure {
    private ExecutionEnvironment qenv;
    private ProcedureReference proc;
    public SimpleProcInvoke() { }
    /**
     * This is called once just after constructing the class. The
     * environment contains methods used to interact with the server.
     */
    public void initialize(ExecutionEnvironment qenv) {
        this.qenv = qenv;
    }
    /**
     * Called during introspection to get the description of the input
     * and output parameters. Should not return null.
     */
    public ParameterInfo[] getParameterInfo() {
        return new ParameterInfo[] {
            new ParameterInfo("arg1", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("operator", Types.VARCHAR, DIRECTION_IN),
            new ParameterInfo("arg2", Types.INTEGER, DIRECTION_IN),
            new ParameterInfo("result", Types.INTEGER, DIRECTION_OUT),
        };
    }
    /**
     * Called to invoke the stored procedure. Will only be called a
     * single time per instance. Can throw CustomProcedureException or
     */
```
public void invoke(Object[] inputValues) throws CustomProcedureException, SQLException {
    proc = qenv.lookupProcedure("/services/databases/tutorial/expr");
    proc.invoke(inputValues);
}

/**
 * Called to retrieve the number of rows that were inserted,
 * updated, or deleted during the execution of the procedure. A
 * return value of -1 indicates that the number of affected rows is
 * unknown. Can throw CustomProcedureException or SQLException if
 * there is an error when getting the number of affected rows.
 */
public int getNumAffectedRows() {
    return 0;
}

/**
 * Called to retrieve the output values. The returned objects
 * should obey the Java to SQL typing conventions as defined in the
 * table above. Output cursors can be returned as either
 * CustomCursor or java.sql.ResultSet. Can throw
 * CustomProcedureException or SQLException if there is an error
 * when getting the output values. Should not return null.
 */
public Object[] getOutputValues() throws CustomProcedureException, SQLException {
    return proc.getOutputValues();
}

/**
 * Called when the procedure reference is no longer needed. Close
 * may be called without retrieving any of the output values (such
 * as cursors) or even invoking, so this needs to do any remaining
 * cleanup. Close may be called concurrently with any other call
 * such as "invoke" or "getOutputValues". In this case, any pending
 * methods should immediately throw a CustomProcedureException.
 */
public void close() throws CustomProcedureException, SQLException {
    if (proc != null)
Examples

Chapter 6 JAVA APIs for Custom Procedures

```java
proc.close();
}

// Introspection methods

/**
 * Called during introspection to get the short name of the stored
 * procedure. This name may be overridden during configuration.
 * Should not return null.
 */
public String getName() {
    return "SimpleProcInvoke";
}
/**
 * Called during introspection to get the description of the stored
 * procedure. Should not return null.
 */
public String getDescription() {
    return "This procedure invokes another procedure.";
}

// Transaction methods

/**
 * Returns true if the custom procedure uses transactions. If this
 * method returns false then commit and rollback will not be called.
 */
public boolean canCommit() {
    return false;
}
/**
 * Commit any open transactions.
 */
public void commit() {
}
/**
 * Rollback any open transactions.
 */
public void rollback() {
}
```
/*
 * Returns true if the transaction can be compensated.
 */

public boolean canCompensate() {
    return false;
}

/*
 * Compensate any committed transactions (if supported).
 */

public void compensate(ExecutionEnvironment qenv) { }

Chapter 7

Composite System Tables

This chapter describes the Composite system tables made accessible to users with administrative rights.

The Modeler resource tree displays this metadata stored by the Composite Server as a set of system tables within the system database parent node.

Composite users and JDBC/ODBC applications with the appropriate permissions may select system table data but the rights and privileges to change data present in the system tables are locked to discourage changes that could compromise functionality and performance.

Composite Software reserves the right to change the system tables with new releases.

Composite Studio may be used to view system table data. After opening the system table simply execute “Show Contents”:

The data includes selected metadata of resources defined for use by client applications with tables such as ALL_COLUMNS, ALL_RESOURCES, ALL_PROCEDURES, and ALL_WSDL_OPERATIONS.
System tables are like relational data source tables except that even Composite Studio users are limited to executing SQL SELECT statements on these tables.

The following system tables exist in Composite:

- Table: ALL_CATALOGS
- Table: ALL_COLUMNS
- Table: ALL_DATASOURCES
- Table: ALL_DOMAINS
- Table: ALL_FOREIGN_KEYS
- Table: ALL_GROUPS
- Table: ALL_INDEXES
- Table: ALL_PARAMETERS
- Table: ALL_PROCEDURES
- Table: ALL_RESOURCES
- Table: ALL_SCHEMAS
- Table: ALL_TABLES
- Table: ALL_USERS
- Table: ALL_WSDL_OPERATIONS
- Table: LOG_DISK
- Table: LOG_EVENTS
- Table: LOG_IO
- Table: LOG_MEMORY
- Table: SYS_CACHES
- Table: SYS_CLUSTER
- Table: SYS_DATASOURCES
- Table: SYS_REQUESTS
- Table: SYS_SESSIONS
- Table: SYS_STATISTICS
- Table: SYS_TRANSACTIONS
- Table: SYS_TRIGGERS
- Table: TRANSACTION_LOG

The following section describes the schema of each of these system tables.

Rights are not directly involved with these tables, but having the rights to Read All Resources or Modify All Resources will result in having effective privileges on all resources.
Table: ALL_CATALOGS

This table exposes all the catalogs to which the current user has access.

Users can see catalogs for which they have at least one privilege.

Table 76. Schema for the system table ALL_CATALOGS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the catalog</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the catalog</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the catalog</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the user who created/owns the catalog Same as USER_ID in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the user who created/owns the catalog Same as USERNAME in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>

Table: ALL_COLUMNS

This table exposes all the columns in all the tables in all the data sources to which the current user has access.
<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the column</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the column</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>String representation of the data type</td>
</tr>
<tr>
<td>ORDINAL_POSITION</td>
<td>INTEGER</td>
<td></td>
<td>Position of this column in relation to other columns in the same table</td>
</tr>
<tr>
<td>JDBC_DATA_TYPE</td>
<td>SMALLINT</td>
<td></td>
<td>JDBC/ODBC data types. For JDBC data types refer to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><a href="http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Types.html">http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Types.html</a></td>
</tr>
<tr>
<td>COLUMN_LENGTH</td>
<td>INTEGER</td>
<td>Yes</td>
<td>For CHAR or VARCHAR columns max length is allowed. For DECIMAL or NUMERIC columns, the total number of digits is the column length value. If it is none of the types named above, then the value is NULL.</td>
</tr>
<tr>
<td>COLUMNPRECISION</td>
<td>INTEGER</td>
<td>Yes</td>
<td>If it is a DECIMAL or NUMERIC data type, then it is the number of digits. If it is not a DECIMAL or NUMERIC data type, then the value is NULL.</td>
</tr>
<tr>
<td>COLUMN_SCALE</td>
<td>INTEGER</td>
<td>Yes</td>
<td>10 for all numeric data types</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 for all non-numeric</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SMALLINT</td>
<td></td>
<td>Indicates whether the column is nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 if NULL is not allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 if NULL is allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 if it is unknown</td>
</tr>
<tr>
<td>LB_NULLABLE</td>
<td>VARCHAR</td>
<td></td>
<td>Indicates whether the column is nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Yes if it is nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NO if it is not nullable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Blank string is returned if it is not known</td>
</tr>
<tr>
<td>TABLE_ID</td>
<td>INTEGER</td>
<td></td>
<td>FK TABLE_ID &quot;Table ALL_TABLES&quot; on page 352</td>
</tr>
</tbody>
</table>
### Table 77. Schema for the system table `ALL_COLUMNS`

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TABLE_NAME</code></td>
<td>VARCHAR</td>
<td></td>
<td>See <code>TABLE_NAME</code> in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td><code>SCHEMA_ID</code></td>
<td>INTEGER</td>
<td>Yes</td>
<td>See <code>SCHEMA_ID</code> in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td><code>SCHEMA_NAME</code></td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See <code>SCHEMA_NAME</code> in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td><code>CATALOG_ID</code></td>
<td>INTEGER</td>
<td>Yes</td>
<td>See <code>CATALOG_ID</code> in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td><code>CATALOG_NAME</code></td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See <code>CATALOG_NAME</code> in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td><code>DATASOURCE_ID</code></td>
<td>INTEGER</td>
<td></td>
<td>See <code>DATASOURCE_ID</code> in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td><code>DATASOURCE_NAME</code></td>
<td>VARCHAR</td>
<td></td>
<td>See <code>DATASOURCE_NAME</code> in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td><code>ANNOTATION</code></td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the column</td>
</tr>
<tr>
<td><code>OWNER_ID</code></td>
<td>INTEGER</td>
<td></td>
<td>Identifiers for the user who created/owns the column Same as <code>USER_ID</code> in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td><code>OWNER</code></td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person that owns/created the data source Same as <code>USERNAME</code> in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td><code>PARENT_PATH</code></td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
### Table: ALL_DATASOURCES

This table exposes all the data sources to which the current user has access. Users can see those data sources for which they have at least one privilege.

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the data source</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the data source</td>
</tr>
<tr>
<td>DATASOURCE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>Data type of the data source. The number and variety of supported data source types are growing with each release.</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the data source</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the user who created/owns the data source</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person that owns/created the data source</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>

Same as USER_ID in "Table: ALL_USERS" on page 353.
Same as USERNAME in "Table: ALL_USERS" on page 353.
Table: ALL_DOMAINS

This table exposes all the domains that are added to the Composite Server. The default domain is composite which is installed during product installation.

Users can see their own domain and the domain of any group they are a member of. In addition, users with the READ_ALL_USERS right can see all domains.

Table 79. Schema for the system table ALL_DOMAINS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMAIN_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the domain.</td>
</tr>
<tr>
<td>DOMAIN_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOMAIN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the domain</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the domain</td>
</tr>
</tbody>
</table>

Table: ALL_FOREIGN_KEYS

ALL_FOREIGN_KEYS exposes all foreign keys discovered on all the tables in all the data sources for which the current user has access privileges.

Users can see foreign keys on tables for which they have at least one privilege.

Table 80. Schema for the system table ALL_FOREIGN_KEYS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FK_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the foreign key</td>
</tr>
<tr>
<td>FK_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the foreign key</td>
</tr>
<tr>
<td>ORDINAL_POSITION</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the foreign key column in relation to other columns in the same foreign key table</td>
</tr>
<tr>
<td>FK_COLUMN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the foreign key column</td>
</tr>
</tbody>
</table>
### Table 80. Schema for the system table ALL_FOREIGN_KEYS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FK_TABLE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See TABLE_ID in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>FK_TABLE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See TABLE_NAME in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>FK_SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>FK_SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>FK_CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 351</td>
</tr>
<tr>
<td>FK_CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 351</td>
</tr>
<tr>
<td>FK_DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 351</td>
</tr>
<tr>
<td>FK_DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 351</td>
</tr>
<tr>
<td>PK_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the parent key name</td>
</tr>
<tr>
<td>PK_COLUMN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the parent key column</td>
</tr>
<tr>
<td>PK_TABLE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See TABLE_ID in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>PK_TABLE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See TABLE_NAME in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>PK_SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>PK_SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>PK_CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 351</td>
</tr>
<tr>
<td>PK_CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 351</td>
</tr>
<tr>
<td>PK_DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 351</td>
</tr>
</tbody>
</table>
Table: ALL_GROUPS

This table exposes all the groups that are added to Composite Server.

Users can see groups in which they are a member. Users with the READ_ALL_USERS right can see all groups.

Table 80. Schema for the system table ALL_FOREIGN_KEYS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK_DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the user who created/owns the foreign key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USER_ID in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the owner/creator of the foreign key</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USERNAME in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>

Table: ALL_GROUPS

This table exposes all the groups that are added to Composite Server.

Users can see groups in which they are a member. Users with the READ_ALL_USERS right can see all groups.

Table 81. Schema for the system table ALL_GROUPS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the group</td>
</tr>
<tr>
<td>GROUP_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the group</td>
</tr>
<tr>
<td>DOMAIN_ID</td>
<td>INTEGER</td>
<td></td>
<td>Unique domain identifier</td>
</tr>
<tr>
<td>DOMAIN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the domain</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Group description</td>
</tr>
</tbody>
</table>
Table: ALL_INDEXES

This table exposes all the indexes on all the tables in all the data sources to which the current user has access.

Table 82. Schema for the system table ALL_INDEXES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the index</td>
</tr>
<tr>
<td>INDEX_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the index</td>
</tr>
<tr>
<td>INDEX_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>Type of the index, whether primary key or other</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the indexed column</td>
</tr>
<tr>
<td>ORDINAL_POSITION</td>
<td>SMALLINT</td>
<td></td>
<td>Position of the indexed column in relation to other columns in the same index</td>
</tr>
<tr>
<td>SORT_ORDER</td>
<td>VARCHAR</td>
<td></td>
<td>Sort order, whether A (for ascending) or D (for descending)</td>
</tr>
<tr>
<td>TABLE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See TABLE_ID in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See TABLE_NAME in “Table: ALL_TABLES” on page 352</td>
</tr>
<tr>
<td>SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>IS_UNIQUE</td>
<td>TINYINT</td>
<td></td>
<td>Indicates whether the index returns unique values</td>
</tr>
<tr>
<td>IS_PRIMARY_KEY</td>
<td>TINYINT</td>
<td></td>
<td>Indicates whether the index is a primary index</td>
</tr>
</tbody>
</table>
Table: ALL_INDEXES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier for the user who created/owns the index</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person that owns/created the index</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>

Table: ALL_PARAMETERS

This table exposes all the parameters that are used in all the procedures to which the current user has access.

Users can see procedures for which they have at least one privilege.

Table 83. Schema for the system table ALL_PARAMETERS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the parameter</td>
</tr>
<tr>
<td>PARAMETER_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the parameter</td>
</tr>
<tr>
<td>DATA_TYPE</td>
<td>CHAR</td>
<td></td>
<td>String representation of the data type.</td>
</tr>
<tr>
<td>DIRECTION</td>
<td>SMALLINT</td>
<td></td>
<td>Type of the parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 means Unknown</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 means IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 means IN &amp; OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 means RESULT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 means OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 means RETURN</td>
</tr>
<tr>
<td>ORIGINAL_POSITION</td>
<td>INTEGER</td>
<td>Yes</td>
<td>Position of the parameter in relation to other parameters in the same procedure.</td>
</tr>
</tbody>
</table>
Table 83. Schema for the system table ALL_PARAMETERS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC_DATA_TYPE</td>
<td>SMALLINT</td>
<td></td>
<td>See JDBC_DATA_TYPE in “Table: ALL_COLUMNS” on page 338</td>
</tr>
<tr>
<td>PARAMETER_LENGTH</td>
<td>INTEGER</td>
<td>Yes</td>
<td>If this is a CHAR or VARCHAR, the length is the maximum length allowed; otherwise, it is NULL.</td>
</tr>
<tr>
<td>PARAMETER_PRECISION</td>
<td>INTEGER</td>
<td>Yes</td>
<td>Value is the number of digits for DECIMAL or NUMERIC data types. If the data type is not DECIMAL or NUMERIC, it is NULL.</td>
</tr>
<tr>
<td>PARAMETER_SCALE</td>
<td>INTEGER</td>
<td>Yes</td>
<td>For a DECIMAL or NUMERIC data type, it is the number of digits. If the data type is not DECIMAL or NUMERIC, it is NULL.</td>
</tr>
<tr>
<td>PARAMETER_RADIX</td>
<td>INTEGER</td>
<td>Yes</td>
<td>Parameter_Radix value is “10” for all numeric data types. For non-numeric data types, it is NULL.</td>
</tr>
<tr>
<td>NULLABLE</td>
<td>SMALLINT</td>
<td></td>
<td>Indicates whether the column is nullable. 0 if NULL is not allowed, 1 if NULL is allowed, 2 if it is unknown.</td>
</tr>
<tr>
<td>IS_NULLABLE</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Indicates whether the column is nullable. YES if it is nullable, NO if it is not nullable, and a blank string is returned if it is not known.</td>
</tr>
<tr>
<td>PROCEDURE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See PROCEDURE_ID in “Table: ALL_PROCEDURES” on page 348</td>
</tr>
<tr>
<td>PROCEDURE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See PROCEDURE_NAME in “Table: ALL_PROCEDURES” on page 348</td>
</tr>
<tr>
<td>SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 338</td>
</tr>
</tbody>
</table>
Table 83. Schema for the system table ALL_PARAMETERS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the parameter</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the person who created/owns the stored procedure in which the parameter is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USER_ID in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person who created/owns the procedure in which the parameter is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USERNNAME in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>

Table: ALL_PROCEDURES

This table exposes all the procedures to which the current user has access.

Users can see procedures for which they have at least one privilege.

Table 84. Schema for the system table ALL_PROCEDURES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the procedure</td>
</tr>
<tr>
<td>PROCEDURE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the procedure</td>
</tr>
</tbody>
</table>
### Table 84. Schema for the system table ALLPROCEDURES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE_TYPE</td>
<td>SMALLINT</td>
<td></td>
<td>Procedure type with possible values: 1, 2, 3, 4, or 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 denotes a relational data source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 denotes a WSDL type of data source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 denotes a flat file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 denotes the Workspace</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 denotes an LDAP data source</td>
</tr>
<tr>
<td>SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td></td>
<td>Annotation for the procedure</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the person who created/owns the procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USER_ID in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person who created/owns the procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Same as USERNAME in “Table: ALL USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
**Table: ALL_RESOURCES**

This table exposes all Composite resources to which the current user has access.

The ALL_RESOURCES table requires the Access Tools right in addition to having at least one privilege of some sort.

**Table 85. Schema for the system table ALL_RESOURCES**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the resource.</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the resource</td>
</tr>
<tr>
<td>RESOURCE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>Type of the resource</td>
</tr>
<tr>
<td>ANNNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the resource</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Definition of the resource. Applicable only to certain resources such as SQL Scripts, packaged queries, XSLT-based transformations LONGVARCHAR (2147483647)</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the user who created/owns the data source Same as USER_ID in &quot;Table: ALL_USERS&quot; on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>User name of the person that owns/created the data source Same as USERNAME in &quot;Table: ALL_USERS&quot; on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
Table: **ALL_SCHEMAS**

This table exposes all the schemas to which the current user has access.

Table 86. Schema for the system table **ALL_SCHEMAS**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the schema</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the schema</td>
</tr>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the schema</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the user who created/owns the schema</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>Same as OWNER_ID in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
Table: **ALL_TABLES**

This table exposes all the tables to which the current user has access. Users can see tables for which they have at least one privilege.

**Table 87. Schema for the system table ALL_TABLES**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the table</td>
</tr>
<tr>
<td>TABLE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the table</td>
</tr>
<tr>
<td>TABLE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>Data type of the table</td>
</tr>
<tr>
<td>SCHEMA_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See SCHEMA_ID in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>SCHEMA_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See SCHEMA_NAME in “Table: ALL_SCHEMAS” on page 351</td>
</tr>
<tr>
<td>CATALOG_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>See CATALOG_ID in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>CATALOG_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>See CATALOG_NAME in “Table: ALL_CATALOGS” on page 338</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DATASOURCE_ID in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in “Table: ALL_DATASOURCES” on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the table</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Identifier of the person who created/owns the table: USER_ID in “Table: ALL_USERS” on page 353 is the same.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the person who created/owns the table. Same as USERNAME in “Table: ALL_USERS” on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
Table: ALL_USERS

This table exposes all the users in all the domains in the Composite Server.

Users can see their own user row. In addition, users with the READ_ALL_USERS right can see all users.

Table 88. Schema for the system table ALL_USERS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the user</td>
</tr>
<tr>
<td>USRNAME</td>
<td>VARCHAR</td>
<td></td>
<td>Login name of the user</td>
</tr>
<tr>
<td>DOMAIN_ID</td>
<td>INTEGER</td>
<td></td>
<td>See DOMAIN_ID in &quot;Table: ALL_DOMAINS&quot; on page 342</td>
</tr>
<tr>
<td>DOMAIN_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DOMAIN_NAME in &quot;Table: ALL_DOMAINS&quot; on page 342</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the user</td>
</tr>
</tbody>
</table>
**Table: ALL_WSDL_OPERATIONS**

This table exposes all the WSDL operations (of Web services and WSDL data sources) to which the current user has access.

Users can see WSDL operations for which they have at least one privilege.

**Table 89. Schema for the system table ALL_WSDL_OPERATIONS**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATION_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the operation.</td>
</tr>
<tr>
<td>OPERATION_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>Name of the operation</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>Primary key identifier of the data source. Refers to DATASOURCE_ID in &quot;Table: ALL_DATASOURCES&quot;</td>
</tr>
<tr>
<td>DATASOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>See DATASOURCE_NAME in &quot;Table: ALL_DATASOURCES&quot; on page 341</td>
</tr>
<tr>
<td>ANNOTATION</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>Annotation for the operation</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Same as USER_ID in &quot;Table: ALL_USERS&quot; on page 353</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>Same as USERID in &quot;Table: ALL_USERS&quot; on page 353</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the parent container</td>
</tr>
</tbody>
</table>
Table: LOG_DISK

The LOG_DISK system table exposes disk space log available on the server.
Users will see no rows unless they have the ACCESS_TOOLS right.

Table 90. Schema for the system table LOG_DISK

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time when the data was logged.</td>
</tr>
<tr>
<td>CONF_DISK_SIZE</td>
<td>BIGINT</td>
<td></td>
<td>The size of the disk where &quot;conf&quot; is located</td>
</tr>
<tr>
<td>CONF_DISK_USED</td>
<td>BIGINT</td>
<td></td>
<td>The amount of space used on the disk.</td>
</tr>
<tr>
<td>TMP_DISK_SIZE</td>
<td>BIGINT</td>
<td></td>
<td>The size of the disk where &quot;tmp&quot; is located</td>
</tr>
<tr>
<td>TMP_DISK_USED</td>
<td>BIGINT</td>
<td></td>
<td>The amount of space used on the disk.</td>
</tr>
<tr>
<td>LOG_DISK_SIZE</td>
<td>BIGINT</td>
<td></td>
<td>The size of the disk where &quot;logs&quot; is located</td>
</tr>
<tr>
<td>LOG_DISK_USED</td>
<td>BIGINT</td>
<td></td>
<td>The amount of space used on the disk.</td>
</tr>
</tbody>
</table>

Table: LOG_EVENTS

The LOG_EVENTS system table enables views of events produced by the server.
Users will not have access to rows unless they have ACCESS_TOOLS and READ_ALL_STATUS rights.

Table 91. Schema for the system table LOG_EVENTS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_ID</td>
<td>BIGINT</td>
<td></td>
<td>The unique ID for this event</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>BIGINT</td>
<td></td>
<td>The ID of the parent of this event. Same as the EVENT_ID if it has no parent</td>
</tr>
<tr>
<td>TYPE_ID</td>
<td>INTEGER</td>
<td></td>
<td>The ID that identifies the type of event that occurred</td>
</tr>
</tbody>
</table>
The LOG_IO table holds data on the I/O produced on the server. Users will not have access to rows unless they have the ACCESS_TOOLS right.

**Table: LOG_IO**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_TIME</td>
<td>TIMESTAMP</td>
<td>NULL</td>
<td>The time when the data was logged.</td>
</tr>
<tr>
<td>FROM_CLIENTS</td>
<td>BIGINT</td>
<td>NULL</td>
<td>Estimated number of bytes sent by clients to the server.</td>
</tr>
<tr>
<td>TO_CLIENTS</td>
<td>BIGINT</td>
<td>NULL</td>
<td>Estimated number of bytes sent by the server to clients.</td>
</tr>
<tr>
<td>FROM_DATASOURCES</td>
<td>BIGINT</td>
<td>NULL</td>
<td>Estimated number of bytes sent by data sources to the server.</td>
</tr>
<tr>
<td>TO_DATASOURCES</td>
<td>BIGINT</td>
<td>NULL</td>
<td>Estimated number of bytes sent by the server to data sources.</td>
</tr>
</tbody>
</table>
### Table: LOG_MEMORY

This table exposes the log of memory available in the server.

Users will see no rows unless they have ACCESS_TOOLS right.

#### Table 93. Schema for the system table LOG_MEMORY

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time when the data was logged.</td>
</tr>
<tr>
<td>MEMORY_BYTES</td>
<td>BIGINT</td>
<td></td>
<td>The amount of Java Heap memory used.</td>
</tr>
<tr>
<td>MEMORY_MAX</td>
<td>BIGINT</td>
<td></td>
<td>The maximum amount of Java Heap memory available.</td>
</tr>
<tr>
<td>MEMORY_BYTES</td>
<td>BIGINT</td>
<td></td>
<td>The amount of managed memory used.</td>
</tr>
<tr>
<td>MANAGED_MAX</td>
<td>BIGINT</td>
<td></td>
<td>The maximum amount of managed memory available.</td>
</tr>
</tbody>
</table>

### Table: SYS_CACHES

This table provides a list of all cached resources and their current status.

Users do not have access to rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all resources they have READ privilege to. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

#### Table 94. Schema for the system table SYS_CACHES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>The cached resource's ID.</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>The cached resource's name.</td>
</tr>
<tr>
<td>RESOURCE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The cached resource's type. Can be 'TABLE' or 'PROCEDURE'.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The cached resource's owner ID.</td>
</tr>
</tbody>
</table>
Table 94. Schema for the system table SYS_CACHES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The cached resource's owner name.</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>The path to the cached resource.</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>The status of the cache. Can be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'DISABLED' - The cache is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'NOT_LOADED' - The cache is enabled, but not loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'UP' - The cache is enabled and loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'STALE' - The cache is enabled and loaded, but the data has expired</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'ERROR' - The cache failed its most recent attempt to load</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'CONFIG ERROR' - The cache is not configured properly.</td>
</tr>
<tr>
<td>VARIANT</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>For procedures, the parameter value or a comma separated list of parameter values submitted for generation of the cache. Variant is NULL for table views. It is NULL for a procedure if no variants are being tracked.</td>
</tr>
<tr>
<td>LAST_REFRESH_END</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time the most recent refresh finished</td>
</tr>
<tr>
<td>LAST_SUCCESS_END</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the most recent successful refresh finished</td>
</tr>
<tr>
<td>LAST_FAIL_END</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the most recent failed refresh finished</td>
</tr>
<tr>
<td>LAST_ACCESS</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the cache was most recently read from</td>
</tr>
<tr>
<td>LAST_SUCCESS_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds the most recent successful refresh took to complete.</td>
</tr>
<tr>
<td>LAST_FAIL_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds the most recent failed refresh took to complete.</td>
</tr>
<tr>
<td>NUM_SUCCESS</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the cache was successfully refreshed since the server was started.</td>
</tr>
<tr>
<td>NUM_FAIL</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the cache failed to refresh since the server was started.</td>
</tr>
<tr>
<td>NUM_ACCESS</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the cache was accessed for read since the server was started.</td>
</tr>
<tr>
<td>STORAGE_USED</td>
<td>BIGINT</td>
<td></td>
<td>The approximate byte size of the cache data.</td>
</tr>
</tbody>
</table>
The SYS_CLUSTER system table provides data on cluster status. There is one row for each server in the cluster, including the current server (although contact info is NULL for the current server).

Users do not have access to data unless they have ACCESS_TOOLS and READ_ALL_STATUS rights.

Refer to the Composite Active Cluster Administration Guide for more information on the SYS_CLUSTER table.

Table: SYS_CLUSTER

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MESSAGE</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>A failure message if the cache is in an error state. NULL if there is no message.</td>
</tr>
<tr>
<td>CURRENT_REFRESH_START</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the current in-progress refresh started. NULL if not currently refreshing.</td>
</tr>
<tr>
<td>CURRENT_DURATION</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The approximate byte size of the cache data currently being refreshed. NULL if not currently refreshing.</td>
</tr>
<tr>
<td>CURRENT_STORAGE</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The reason the cache is refreshing. NULL if not currently refreshing. Can be 'MANUAL', 'SCHEDULED', 'EXPIRED', or 'ON_DEMAND'.</td>
</tr>
</tbody>
</table>
Table: SYS_DATASOURCES

The SYS_DATASOURCES system table provides a list of all data sources and their current status.

Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all resources they have READ privilege to. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

Table 95. Schema for the system table SYS_DATASOURCES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>The data source's resource ID.</td>
</tr>
<tr>
<td>SOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>The data source's resource name</td>
</tr>
<tr>
<td>SOURCE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The data source's data source type. For example, 'MySql'.</td>
</tr>
<tr>
<td>SOURCE_CATEGORY</td>
<td>VARCHAR</td>
<td></td>
<td>The data source category. Values can be 'RELATIONAL', 'FILE', and 'SERVICE'.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The data source resource owner ID.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The data source resource owner name.</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>The path of the data source resource. Can be NULL for system owned data sources.</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>Data source current status</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DISABLED - Data source disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UP - Data source enabled and running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DOWN - Data source down when last tested.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NOT_TESTED - Data source not tested, status unknown.</td>
</tr>
<tr>
<td>NUM_REQUESTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of requests processed since the server started.</td>
</tr>
<tr>
<td>MAX_CONN</td>
<td>INTEGER</td>
<td></td>
<td>The maximum size of the data source's connection pool.</td>
</tr>
<tr>
<td>NUM_CURRENT_CONN</td>
<td>INTEGER</td>
<td></td>
<td>The current size of the data source's connection pool.</td>
</tr>
<tr>
<td>NUM_IN_USE_CONN</td>
<td>INTEGER</td>
<td></td>
<td>The number of data source connections currently in use.</td>
</tr>
</tbody>
</table>
Table 95. Schema for the system table SYS_DATASOURCES

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM_LOGINS</td>
<td>INTEGER</td>
<td></td>
<td>The number of times new connections were opened since the server started.</td>
</tr>
<tr>
<td>NUM_LOGOUTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of times connections were closed since the server started.</td>
</tr>
<tr>
<td>BYTES_TO</td>
<td>BIGINT</td>
<td></td>
<td>The estimated number of bytes sent to the data source since the server started.</td>
</tr>
<tr>
<td>BYTES_FROM</td>
<td>BIGINT</td>
<td></td>
<td>The estimated number of bytes retrieved from the data source since the server started.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>A message about the data source. NULL if no message is available.</td>
</tr>
</tbody>
</table>
Table: SYS_REQUESTS

This table provides a list of current and recent requests and their current status.
Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all requests they own. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

Table 96. Schema for the system table SYS_REQUESTS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUEST_ID</td>
<td>BIGINT</td>
<td></td>
<td>The request's ID.</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The parent request's ID. NULL if there is no parent request.</td>
</tr>
<tr>
<td>SESSION_ID</td>
<td>BIGINT</td>
<td></td>
<td>The request's session ID.</td>
</tr>
<tr>
<td>TRANSACTION_ID</td>
<td>BIGINT</td>
<td></td>
<td>The request's transaction ID.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The request session's user ID.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The request session's user name.</td>
</tr>
<tr>
<td>REQUEST_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The request type. For example, 'SQL' or 'SQL Script'.</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>The request status which may be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'STARTED' - The request is in the process of starting. This status is usually very brief.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'WAITING' - The request is waiting for enough system resources in order to start running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'RUNNING' - The request is currently executing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'READY' - The request has completed execution and results are available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'CLOSING' - The request is in the process of closing. This status is usually very brief.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'SUCCESS' - The request was completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'FAILED' - The request failed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'TERMINATED' - The request was terminated.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>VARCHAR</td>
<td></td>
<td>The request's source or a description of what was called.</td>
</tr>
<tr>
<td>START_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time when the request started.</td>
</tr>
<tr>
<td>END_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time when the request ended. NULL if it is still running.</td>
</tr>
<tr>
<td>TOTAL_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds the request was executed.</td>
</tr>
</tbody>
</table>
### Table 96. Schema for the system table SYS_REQUESTS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds of server-side time during the request's execution.</td>
</tr>
<tr>
<td>ROWS_AFFECTED</td>
<td>BIGINT</td>
<td></td>
<td>The number of rows affected by the request. For SQL SELECT statements, this is the number of rows read. For other requests, this is the number of rows modified. A value of -1 indicates that the number is not known.</td>
</tr>
<tr>
<td>MAX_MEMORY</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The maximum amount of memory reserved by the request during execution.</td>
</tr>
<tr>
<td>MAX_DISK</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The maximum amount of disk used by the request during execution.</td>
</tr>
<tr>
<td>CURRENT_MEMORY</td>
<td>BIGINT</td>
<td></td>
<td>The current amount of memory reserved by the request.</td>
</tr>
<tr>
<td>CURRENT_DISK</td>
<td>BIGINT</td>
<td>Yes</td>
<td>The current amount of disk in use by the request.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>A message that is usually set on failure to provide additional information. NULL if no message is available.</td>
</tr>
</tbody>
</table>
### Table: SYS_SESSIONS

This table provides a list of current and recent sessions and their current status. Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all sessions they own. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION_ID</td>
<td>BIGINT</td>
<td></td>
<td>Unique session ID</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The ID of the user logged into this session.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The name of the user logged into this session.</td>
</tr>
<tr>
<td>SESSION_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The session type may be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘HTTP’ - A web services client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘INTERNAL’ - A session started within the server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘JDBC’ - A JDBC client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘ODBC’ - An ODBC client.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘STUDIO’ - The Composite Studio tool.</td>
</tr>
<tr>
<td>SESSION_NAME</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>The name of the session. NULL if not provided by the client.</td>
</tr>
<tr>
<td>HOST</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>The host the client is connecting from. NULL for INTERNAL sessions.</td>
</tr>
<tr>
<td>DATASOURCE_ID</td>
<td>INTEGER</td>
<td>Yes</td>
<td>The data service ID the client is connecting on. NULL if no data service is in use.</td>
</tr>
<tr>
<td>LOGIN_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time the session was started.</td>
</tr>
<tr>
<td>LOGOUT_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time the session was ended. NULL if the session is still active.</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>The session status may be one of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘ACTIVE’ - The session is still active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘CLOSED’ - The session was closed in an orderly fashion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘DISCONNECTED’ - The session was disconnected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘TERMINATED’ - The session was terminated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>‘TIMED_OUT’ - The session was timed out.</td>
</tr>
</tbody>
</table>
Table: SYS_STATISTICS

This table provides a list of current and recent sessions and their current status. Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all resources they have READ privilege to. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds the session has been idle.</td>
</tr>
<tr>
<td>TIMEOUT_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds after which the session will be timed out.</td>
</tr>
<tr>
<td>TOTAL_REQUESTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of requests created on this session.</td>
</tr>
<tr>
<td>ACTIVE_REQUESTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of requests open on this session.</td>
</tr>
<tr>
<td>TOTAL_TRANSACTIONS</td>
<td>INTEGER</td>
<td></td>
<td>The number of transactions created on this session.</td>
</tr>
<tr>
<td>ACTIVE_TRANSACTIONS</td>
<td>INTEGER</td>
<td></td>
<td>The number of transactions open on this session.</td>
</tr>
<tr>
<td>BYTES_TO_CLIENT</td>
<td>BIGINT</td>
<td></td>
<td>The estimated number of bytes sent to the client.</td>
</tr>
<tr>
<td>BYTES_FROM_CLIENT</td>
<td>BIGINT</td>
<td></td>
<td>The estimated number of bytes received from the client.</td>
</tr>
</tbody>
</table>

Table: SYS_STATISTICS

This table provides a list of current and recent sessions and their current status. Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all resources they have READ privilege to. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

Table 97. Schema for the system table SYS_STATISTICS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>The resource’s ID.</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>The resource’s name.</td>
</tr>
<tr>
<td>RESOURCE_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The resource’s type. Can be ‘TABLE’ or ‘DATASOURCE’.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>Owner’s user ID.</td>
</tr>
</tbody>
</table>
Table 97. Schema for the system table SYS_STATISTICS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>Owner name.</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>Path to the resource.</td>
</tr>
<tr>
<td>IS_ENABLED</td>
<td>VARCHAR</td>
<td></td>
<td>Indicates if statistics data will be used. Can be 'true' or 'false'.</td>
</tr>
<tr>
<td>LAST_REFRESH_END</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time the last gather process finished.</td>
</tr>
<tr>
<td>LAST_SUCCESS_END</td>
<td>TIMESTAMP</td>
<td></td>
<td>The last time gather process finished successfully.</td>
</tr>
<tr>
<td>LAST_FAIL_END</td>
<td>TIMESTAMP</td>
<td></td>
<td>The last time gather process finished with an error.</td>
</tr>
<tr>
<td>LAST_SUCCESS_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>Elapsed time (in milliseconds) of the last successful stats gather process.</td>
</tr>
<tr>
<td>LAST_FAIL_DURATION</td>
<td>BIGINT</td>
<td></td>
<td>Elapsed time (in milliseconds) of the last failed stats gather process.</td>
</tr>
<tr>
<td>NUM_SUCCESS</td>
<td>INTEGER</td>
<td></td>
<td>Number of times stats data was successfully refreshed since last server start.</td>
</tr>
<tr>
<td>NUM_FAIL</td>
<td>INTEGER</td>
<td></td>
<td>Number of times stats data failed to refresh since last server start.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>VARCHAR</td>
<td></td>
<td>Informational message that provides additional information for some status types.</td>
</tr>
<tr>
<td>CURRENT_REFRESH_START</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time currently running stats gather process started. NULL if not currently running.</td>
</tr>
<tr>
<td>CURRENT_DURATION</td>
<td>BIGINT</td>
<td>Yes</td>
<td>Elapsed time of currently running stats gather process. NULL if not currently running.</td>
</tr>
</tbody>
</table>
Table: SYS_TRANSACTIONS

This table provides a list of current and recent transactions and their current status.

Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all transactions they own. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

Table 98. Schema for the system table SYS_TRANSACTIONS

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSACTION_ID</td>
<td>BIGINT</td>
<td></td>
<td>The unique id for the transaction to which this log entry applies.</td>
</tr>
<tr>
<td>SESSION_ID</td>
<td>BIGINT</td>
<td></td>
<td>The transaction's session ID.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The ID of the user logged into this session.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The name of the user logged into this session.</td>
</tr>
<tr>
<td>MODE</td>
<td>VARCHAR</td>
<td></td>
<td>The mode of the transaction.</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>Status of the transaction</td>
</tr>
<tr>
<td>START_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The time when the transaction was started.</td>
</tr>
<tr>
<td>END_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time when the transaction completed. NULL if it is still in progress.</td>
</tr>
<tr>
<td>DURATION</td>
<td>BIGINT</td>
<td></td>
<td>The number of milliseconds the transaction was running.</td>
</tr>
<tr>
<td>TOTAL_REQUESTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of requests created in the transaction.</td>
</tr>
<tr>
<td>ACTIVE_REQUESTS</td>
<td>INTEGER</td>
<td></td>
<td>The number of requests active in the transaction.</td>
</tr>
</tbody>
</table>
**Table: SYS_TRIGGERS**

This table provides a list of triggers defined in the system and their current status. Users will see no rows unless they have ACCESS_TOOLS right. If they have this right, they will see rows for all resources they have READ privilege to. In addition, a user with both ACCESS_TOOLS and READ_ALL_STATUS rights can see all rows.

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESOURCE_ID</td>
<td>INTEGER</td>
<td></td>
<td>The trigger’s resource ID.</td>
</tr>
<tr>
<td>RESOURCE_NAME</td>
<td>VARCHAR</td>
<td></td>
<td>The trigger’s resource name.</td>
</tr>
<tr>
<td>OWNER_ID</td>
<td>INTEGER</td>
<td></td>
<td>The trigger resource owner ID.</td>
</tr>
<tr>
<td>OWNER</td>
<td>VARCHAR</td>
<td></td>
<td>The trigger resource owner name.</td>
</tr>
<tr>
<td>PARENT_PATH</td>
<td>VARCHAR</td>
<td></td>
<td>The path of the trigger resource. Field length: 65535</td>
</tr>
<tr>
<td>PARENT_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>The type of the trigger’s parent resource.</td>
</tr>
<tr>
<td>CONDITION_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>“The trigger’s condition type. For example, ‘TIMER’.”</td>
</tr>
<tr>
<td>ACTION_TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>“The trigger’s action type. For example, ‘PROCEDURE’.”</td>
</tr>
<tr>
<td>STATUS</td>
<td>VARCHAR</td>
<td></td>
<td>The trigger’s current status. ‘DISABLED’ - The trigger is disabled. ‘ACTIVE’ - The trigger is enabled.</td>
</tr>
<tr>
<td>LAST_TIME</td>
<td>TIMESTAMP</td>
<td></td>
<td>The most recent time the trigger fired.</td>
</tr>
<tr>
<td>LAST_SUCCESS</td>
<td>TIMESTAMP</td>
<td></td>
<td>The most recent time the trigger succeeded.</td>
</tr>
<tr>
<td>LAST_FAIL</td>
<td>TIMESTAMP</td>
<td></td>
<td>The most recent time the trigger failed.</td>
</tr>
<tr>
<td>NUM_TOTAL</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the trigger has fired.</td>
</tr>
<tr>
<td>NUM_SUCCESS</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the trigger has succeeded.</td>
</tr>
<tr>
<td>NUM_FAIL</td>
<td>INTEGER</td>
<td></td>
<td>The number of times the trigger has failed.</td>
</tr>
<tr>
<td>INITIAL_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the trigger was configured to first start. NULL if not condition type ‘TIMER’.</td>
</tr>
<tr>
<td>NEXT_TIME</td>
<td>TIMESTAMP</td>
<td>Yes</td>
<td>The time the trigger will next fire. NULL if not condition type ‘TIMER’.</td>
</tr>
<tr>
<td>Column</td>
<td>Composite JDBC Data Type</td>
<td>Nullable</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FREQUENCY</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>English description of the frequency of the trigger. NULL if not condition type 'TIMER'.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>VARCHAR</td>
<td>Yes</td>
<td>A message about the trigger status that is often set on failure; NULL if no message is available. Field length: 65535</td>
</tr>
</tbody>
</table>
**Table: TRANSACTION_LOG**

This table provides a read-only view of the transaction log. The transaction log is used to maintain the state on the lifecycle of a transaction. The purpose of the transaction log is to store information about the transaction in case there is a failure while performing the transaction commit. Log data can be used to manually recover data from a transaction failure. Also, in some cases, the log contains information for automatic compensation for a failed transactions when the server is interrupted in the middle of a commit operation. Successful transactions are automatically removed from the log upon completion of the commit or rollback operation. Failed transactions remain in the log.

Table view requires the ACCESS_TOOLS and READ_ALL_STATUS rights.

**Table 99. Schema for the system table TRANSACTION_LOG**

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>VARCHAR</td>
<td></td>
<td>Indicates the type of transaction log entry.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin transaction (manual) -- start a transaction supporting manual recovery</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin transaction (auto) -- start a transaction supporting both manual recovery and automatic compensation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Execute SQL -- execute an SQL statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Add work unit -- add a work unit, where a work unit indicates an insert/update/delete action on a datasource</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>End commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fail commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin rollback</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>End rollback</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fail rollback</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Server restart</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin work unit commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>End work unit commit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Work unit commit failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Work unit commit in doubt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Begin work unit rollback</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>End work unit rollback</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Work unit rollback failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Being work unit compensate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>End work unit compensate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Work unit compensate failure</td>
</tr>
<tr>
<td>SERIAL</td>
<td>BIGINT</td>
<td></td>
<td>Unique serial number for the transaction log entry</td>
</tr>
</tbody>
</table>
### Table 99. Schema for the system table TRANSACTION_LOG

<table>
<thead>
<tr>
<th>Column</th>
<th>Composite JDBC Data Type</th>
<th>Nullable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>BIGINT</td>
<td></td>
<td>The time when the log entry was made in milliseconds.</td>
</tr>
<tr>
<td>TRANSACTION_ID</td>
<td>BIGINT</td>
<td></td>
<td>The unique id for the transaction to which this log entry applies.</td>
</tr>
<tr>
<td>WORK_UNIT_ID</td>
<td>BIGINT</td>
<td>Yes</td>
<td>For work unit entries, this is the unique id for the work unit, otherwise NULL.</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>BLOB</td>
<td>Yes</td>
<td>Contains an SQL statement for Execute SQL and Add work unit, contains the exception message for any of the failure types; otherwise NULL.</td>
</tr>
</tbody>
</table>
Symbols

% (modulo) 87
* (multiply) 87
+ (add) 80
= (equal to) 98
> (greater than) 98
>= (greater than or equal to) 99
|| (concatenate) 17

A

ABS 59
ACOS 60
AND 102
arithmetic operators 80
ASIN 60
ATAN 61
attributes
  current exception 161
cursor 159
AVG 7

B

BEGIN...END
SQL Script 177
BETWEEN 115
BLOB 40, 152, 157
built-in procedures
debugging 246
GenerateEvent 250
GetEnvironment 250
GetProperty 251
information, Info panel 248
Log 248
LogError 249
Pause 253
Print 249
resource management 246, 247
SendEMail 253
SetEnvironment 254
system 246
utility programs 247

C

CALL 177
CARDINALITY 197
CASE 103, 180
CAST 40, 197
CEILING 61
CHAR_LENGTH 21
CHARACTER_LENGTH 21
CHR 16
CLOB 40, 151, 157
CLOSE 182
COALESCE 106
COMMIT 182
compound statement
SQL Script 167
CONCAT 17, 197
condition operators 103
CASE 103
ESCAPE in LIKE 113
IN 108
IS NOT NULL 111
IS NULL 112
LIKE 112
conditional expressions, SQL script 155
CONSTANT 183
contents, of this book vi
convert 39
correlated subquery 139
correlation variable 140
COS 62
COT 62
COUNT 9
CROSS JOIN 116
current exception
attributes 161
CURRENT_DATE 55
CURRENT_TIME 55
CURRENT_TIMESTAMP 55
cursor
attributes 159
declare 184

D

data type

string 5
xml 5
data types
Composite JDBC and data sources 256
SQL script 149
data types, notes 2
date functions
date 54
DAY 55
declaring
virtual columns 128
DEGREES 63
DELETE 116, 202
DISTINCT 15, 117
divide (/ ) 86
documentation set vii

E

ESCAPE 113
EXCEPT 117
EXCEPTION 189
exceptions
SQL Script 174
EXECUTE IMMEDIATE 203
EXISTS 106
EXP 63
EXTEND 197
EXTRACT 56

F

FETCH 204
FIND_INDEX 197
FIRST
keyword, FETCH 205
FLOOR 64
FOR 205
FORMAT
FORMAT_DATE 48
<table>
<thead>
<tr>
<th>G</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSE_DATE 50</td>
<td></td>
</tr>
<tr>
<td>PARSE_TIME 51</td>
<td></td>
</tr>
<tr>
<td>PARSE_TIMESTAMP 51</td>
<td></td>
</tr>
<tr>
<td>FORMAT_DATE 48</td>
<td></td>
</tr>
<tr>
<td>FULL OUTER JOIN 119</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>ABS 59</td>
<td></td>
</tr>
<tr>
<td>ACOS 60</td>
<td></td>
</tr>
<tr>
<td>aggregate 7</td>
<td></td>
</tr>
<tr>
<td>ASIN 60</td>
<td></td>
</tr>
<tr>
<td>ATAN 61</td>
<td></td>
</tr>
<tr>
<td>AVG 7</td>
<td></td>
</tr>
<tr>
<td>CAST 40</td>
<td></td>
</tr>
<tr>
<td>CEILING 61</td>
<td></td>
</tr>
<tr>
<td>CHAR_LENGTH 21</td>
<td></td>
</tr>
<tr>
<td>CHARACTER_LENGTH 21</td>
<td></td>
</tr>
<tr>
<td>CONCAT 17</td>
<td></td>
</tr>
<tr>
<td>convert 39</td>
<td></td>
</tr>
<tr>
<td>COS 62</td>
<td></td>
</tr>
<tr>
<td>COT 62</td>
<td></td>
</tr>
<tr>
<td>COUNT 9</td>
<td></td>
</tr>
<tr>
<td>CURRENT_DATE 55</td>
<td></td>
</tr>
<tr>
<td>CURRENT_TIME 55</td>
<td></td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP 55</td>
<td></td>
</tr>
<tr>
<td>DAY 55</td>
<td></td>
</tr>
<tr>
<td>DEGREES 63</td>
<td></td>
</tr>
<tr>
<td>DISTINCT 15</td>
<td></td>
</tr>
<tr>
<td>EXP 63</td>
<td></td>
</tr>
<tr>
<td>FLOOR 64</td>
<td></td>
</tr>
<tr>
<td>including in SQL 7</td>
<td></td>
</tr>
<tr>
<td>LENGTH 21</td>
<td></td>
</tr>
<tr>
<td>LOG 64</td>
<td></td>
</tr>
<tr>
<td>LOWER 22</td>
<td></td>
</tr>
<tr>
<td>MAX 10</td>
<td></td>
</tr>
<tr>
<td>MIN 12</td>
<td></td>
</tr>
<tr>
<td>MONTH 55</td>
<td></td>
</tr>
<tr>
<td>PI 65</td>
<td></td>
</tr>
<tr>
<td>POWER 65</td>
<td></td>
</tr>
<tr>
<td>RADIANS 66</td>
<td></td>
</tr>
<tr>
<td>REPLACE 25</td>
<td></td>
</tr>
<tr>
<td>ROUND 66</td>
<td></td>
</tr>
<tr>
<td>RTRIM 26</td>
<td></td>
</tr>
<tr>
<td>SIN 68</td>
<td></td>
</tr>
<tr>
<td>SPACE 27</td>
<td></td>
</tr>
<tr>
<td>SQRT 68</td>
<td></td>
</tr>
<tr>
<td>SUBSTRING 28</td>
<td></td>
</tr>
<tr>
<td>SUM 13</td>
<td></td>
</tr>
<tr>
<td>TAN 69</td>
<td></td>
</tr>
<tr>
<td>TO_NUMBER 52</td>
<td></td>
</tr>
<tr>
<td>TO_TIMESTAMP 53</td>
<td></td>
</tr>
<tr>
<td>TRIM 31</td>
<td></td>
</tr>
<tr>
<td>UPPER 32</td>
<td></td>
</tr>
<tr>
<td>YEAR 55</td>
<td></td>
</tr>
</tbody>
</table>

G

GenerateEvent, built-in procedure 250
GetEnvironment, built-in procedure 250
GetProperty, built-in procedure 251
GROUP BY 119

H

HAVING 120

I

identifiers 148
IF 208
implicit cursor 219
IN 108
INNER JOIN 120
INSERT 121, 209
INSERT/UPDATE/DELETE not allowed in views 136
INTERSECT 125
INTERVAL 56, 81, 86, 88, 91 in ABS 59
in CAST 41
in EXTRACT 57
INTERVAL DAY 2, 3
INTERVAL YEAR 2, 4
IS NOT NULL 111
IS NULL 112
ITERATE 210

K
keywords 114
DELETE 116
DISTINCT 117
FULL OUTER JOIN 119
GROUP BY 119
HAVING 120
INNER JOIN 120
INSERT 121
LEFT OUTER JOIN 126
ORDER BY 126
RIGHT OUTER JOIN 128
SELECT 128
UNION ALL 133
UPDATE 134
WHERE 136
WITH 137

L
LEAVE 211
LEFT OUTER JOIN 126
LENGTH 21
LIKE 112
LOG 64
Log, built-in procedure 248
LogError, built-in procedure 249
logical operators 102
AND 102
NOT 102
OR 103
LOOP 212
LOWER 22

M
MAX 10
MIN 12
MINUS 117
MONTH 55

N
namespaces, in XML 74
NEXT
keyword, FETCH 205
NOT 102
NOT IN 111
NULLIF 33
numeric functions
numeric 59

O
OPEN 213
operators
- (negate) 90
- (subtract) 91
% (modulo) 87
* (multiply) 87
+ (add) 80
/ (divide) 86
= (equal to) 98
> (greater than) 98
>= (greater than or equal to) 99
|| (concatenate) 17
arithmetic 80
comparison 97
OR 103
ORDER BY 126
<table>
<thead>
<tr>
<th>Index</th>
<th>P Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>PARSE_DATE</td>
<td>50</td>
</tr>
<tr>
<td>PARSE_TIME</td>
<td>51</td>
</tr>
<tr>
<td>PARSE_TIMESTAMP</td>
<td>51</td>
</tr>
<tr>
<td>PATH</td>
<td>214</td>
</tr>
<tr>
<td>Pause, built-in procedure</td>
<td>253</td>
</tr>
<tr>
<td>PI</td>
<td>65</td>
</tr>
<tr>
<td>PIPE</td>
<td>166</td>
</tr>
<tr>
<td>POWER</td>
<td>65</td>
</tr>
<tr>
<td>Print, built-in procedure</td>
<td>249</td>
</tr>
<tr>
<td>procedures</td>
<td></td>
</tr>
<tr>
<td>built-in, system</td>
<td>246</td>
</tr>
<tr>
<td>structure, SQL Script</td>
<td>163</td>
</tr>
<tr>
<td>user-defined vs. built-in</td>
<td>248</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>RADIANS</td>
<td>66</td>
</tr>
<tr>
<td>RAISE</td>
<td>215</td>
</tr>
<tr>
<td>REPEAT</td>
<td>217</td>
</tr>
<tr>
<td>REPLACE</td>
<td>25</td>
</tr>
<tr>
<td>reserved words</td>
<td>140</td>
</tr>
<tr>
<td>RIGHT OUTER JOIN</td>
<td>128</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>218</td>
</tr>
<tr>
<td>ROUND</td>
<td>66</td>
</tr>
<tr>
<td>RTRIM</td>
<td>26</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>scalar subquery</td>
<td>139</td>
</tr>
<tr>
<td>SCROLL</td>
<td></td>
</tr>
<tr>
<td>keyword, CURSOR</td>
<td>184</td>
</tr>
<tr>
<td>SELECT</td>
<td>128</td>
</tr>
<tr>
<td>SELECT INTO</td>
<td>218</td>
</tr>
<tr>
<td>SEMI-JOIN, to a procedure</td>
<td>131</td>
</tr>
<tr>
<td>SendEMail, built-in procedure</td>
<td>253</td>
</tr>
<tr>
<td>SET</td>
<td>219</td>
</tr>
<tr>
<td>SetEnvironment, built-in procedure</td>
<td>254</td>
</tr>
<tr>
<td>SIN</td>
<td>68</td>
</tr>
<tr>
<td>SPACE</td>
<td>27</td>
</tr>
<tr>
<td>SQL keywords</td>
<td>114</td>
</tr>
<tr>
<td>SQL Script</td>
<td></td>
</tr>
<tr>
<td>overview</td>
<td>146</td>
</tr>
<tr>
<td>SQL support</td>
<td>1</td>
</tr>
<tr>
<td>SQRT</td>
<td>68</td>
</tr>
<tr>
<td>streaming cursor, PIPE</td>
<td>166</td>
</tr>
<tr>
<td>string data type</td>
<td>5</td>
</tr>
<tr>
<td>subquery</td>
<td>138</td>
</tr>
<tr>
<td>correlated</td>
<td>139</td>
</tr>
<tr>
<td>scalar</td>
<td>139</td>
</tr>
<tr>
<td>SUBSTRING</td>
<td>28</td>
</tr>
<tr>
<td>SUM</td>
<td>13</td>
</tr>
<tr>
<td>system</td>
<td></td>
</tr>
<tr>
<td>procedures, built-in</td>
<td>246</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>ALL_CATALOGS</td>
<td>338</td>
</tr>
<tr>
<td>ALL_COLUMNS</td>
<td>338</td>
</tr>
<tr>
<td>SYS_REQUESTS</td>
<td>362</td>
</tr>
<tr>
<td>SYS_SESSIONS</td>
<td>364</td>
</tr>
<tr>
<td>SYS_STATISTICS</td>
<td>365</td>
</tr>
<tr>
<td>SYS_TRANSACTIONS</td>
<td>367</td>
</tr>
<tr>
<td>SYS_TRIGGERS</td>
<td>368</td>
</tr>
<tr>
<td>TRANSACTION_LOG</td>
<td>370</td>
</tr>
<tr>
<td>TAN</td>
<td>69</td>
</tr>
<tr>
<td>Teradata</td>
<td>269</td>
</tr>
<tr>
<td>Teradata Data Types</td>
<td>269</td>
</tr>
<tr>
<td>TO_NUMBER</td>
<td>52</td>
</tr>
<tr>
<td>TO_TIMESTAMP</td>
<td>53</td>
</tr>
<tr>
<td>transactions</td>
<td></td>
</tr>
<tr>
<td>compensating, SQL Script</td>
<td>172</td>
</tr>
<tr>
<td>independent, SQL Script</td>
<td>169</td>
</tr>
<tr>
<td>TRIM</td>
<td>31</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>198</td>
</tr>
<tr>
<td>TYPE</td>
<td>191</td>
</tr>
</tbody>
</table>

376
<table>
<thead>
<tr>
<th>U</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION</td>
<td>YEAR 55</td>
</tr>
<tr>
<td>UNION ALL</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>UPPER</td>
<td></td>
</tr>
<tr>
<td>UTC</td>
<td></td>
</tr>
<tr>
<td>UTC_TO_TIMESTAMP</td>
<td>88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>value expressions, SQL script 154</td>
</tr>
<tr>
<td>variables</td>
</tr>
<tr>
<td>declaring 192</td>
</tr>
<tr>
<td>using 158</td>
</tr>
<tr>
<td>VECTOR 194</td>
</tr>
<tr>
<td>virtual columns 128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE 136</td>
</tr>
<tr>
<td>WHILE 221</td>
</tr>
<tr>
<td>WITH 137</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>xml data type 5</td>
</tr>
<tr>
<td>XMLAGG 15, 69</td>
</tr>
<tr>
<td>XMLATTRIBUTES 69</td>
</tr>
<tr>
<td>XMLCONCAT 71</td>
</tr>
<tr>
<td>XMLDOCUMENT 71</td>
</tr>
<tr>
<td>XMLELEMENT 72</td>
</tr>
<tr>
<td>XMLFOREST 73</td>
</tr>
<tr>
<td>XMLNAMESPACES 74</td>
</tr>
<tr>
<td>XMLTEXT 76</td>
</tr>
<tr>
<td>XPATH 77</td>
</tr>
<tr>
<td>XSLT 79</td>
</tr>
</tbody>
</table>