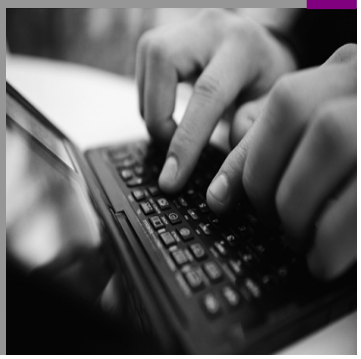
October 2006



How-to Guide

SAP NetWeaver '04

How To... Use Query Functions with the Distributed Query Engine (DQE)

Version 3.00 – April 2006

Applicable Releases:

SAP NetWeaver '04

THE BEST-RUN BUSINESSES RUN SAP

Contents

Query System Features

This booklet is designed to serve as a reference for developers of SAP NetWeaver ’04 portal content. It is meant to supply the technical information necessary for content that retrieves data from back-end applications using the Distributed Query Engine (DQE).

Query Constructs

Supported SQL Constructs

SQL queries start with the SELECT keyword and are often referred to as “SELECT statements” as well. The SAP Distributed Query Engine (DQE) supports most but not all of the standard SQL query constructs. Many of these features can be combined to execute complex queries against the DQE service.

The following list of features gives an overview of the kinds of queries that can be executed using the Distributed Query Engine. Nearly all of these features follow standard SQL syntax and functionality. Please refer to your preferred standard SQL reference for detailed information on how to use these features.

The ordering of the clauses in a query are as follows: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY, and OPTION. All of these clauses are optional except SELECT and FROM, which are required.

SELECT Clause Features

- SELECT *
- SELECT table.*
- SELECT expression, …
- SELECT expression AS name
- SELECT DISTINCT expression
- SELECT COUNT(*)
- SELECT COUNT(expression), COUNT(DISTINCT expression)
- SELECT SUM(expression), SUM (DISTINCT expression)
- SELECT AVG(expression), AVG (DISTINCT expression)
- SELECT MIN(expression), MIN (DISTINCT expression)
- SELECT MAX(expression), MAX (DISTINCT expression)

FROM Clause Features

- FROM table
- FROM table AS name
- FROM table AS name1, table AS name2
- FROM table1 JOIN table2 ON join-criteria
- FROM table1 CROSS JOIN table2
- FROM table1 RIGHT OUTER JOIN table2 ON join-criteria
- FROM table1 LEFT OUTER JOIN table2 ON join-criteria
- FROM table1 FULL OUTER JOIN table2 ON join-criteria

- FROM (table1 JOIN table2 ON join-criteria) JOIN table3 ON join-criteria
- FROM (SELECT …) AS name

WHERE Clause Features

- WHERE expression1 = expression2
- WHERE expression1 <> expression2
- WHERE expression1 < expression2
- WHERE expression1 <= expression2
- WHERE expression1 > expression2
- WHERE expression1 >= expression2
- WHERE expression1 LIKE expression2
- WHERE expression1 LIKE expression2 ESCAPE character
- WHERE expression1 IN (expression2, …)
- WHERE expression1 IN (SELECT expression2…)
- WHERE expression1 IS NULL
- WHERE NOT criteria
- WHERE criteria1 AND criteria2 AND …
- WHERE criteria1 OR criteria2 OR …
- WHERE (criteria1 AND criteria2) OR NOT criteria3

GROUP BY Clause Feature

- GROUP BY column1, …

HAVING Clause Features

- Same as WHERE clause features
- Expressions may additionally contain aggregate functions:
    - COUNT
    - AVG
    - SUM
    - MIN
    - MAX

ORDER BY Clause Features

- ORDER BY column1, column2
- ORDER BY column1 ASC, column2 DESC

Set Operations

- SELECT … UNION SELECT …
- SELECT … UNION ALL SELECT …
- SELECT … UNION SELECT … ORDER BY …

OPTION Clause Feature

- OPTION SHOWPLAN – returns query plan description with results for debugging
- OPTION MAKEDEP physicalTable1 physicalTable2 – specifies physical tables that should be made dependent in the join

UNSUPPORTED SQL CONSTRUCTS

Some SQL query constructs not supported in the Distributed Query Engine include:

- Correlated subqueries, scalar subqueries in SELECT clause and WHERE criteria, and some other forms of subqueries
- EXISTS
- ANY, ALL, SOME, UNIQUE quantified subqueries
- INTERSECT, EXCEPT set operations
- ORDER BY columns specified by column position
- BETWEEN criteria

This list is not a complete or definitive list of the items within the SQL-92 specification that the SAP Distributed Query Engine does not support. This booklet, however, describes the portions of the SQL-92 specification that the DQE does support.

METADATA IDENTIFIERS

Metadata identifiers have the following forms:

- Tables: <Default_System_Alias>.Table
- Columns: <<Default_System_Alias>.Table.>Column
- Aliases: Alias

In addition the following rules govern the specifics:

Metadata Tables

When referring to metadata Tables, the identifier should be a fully qualified name that includes hierarchical metadata objects separated by periods (.).

This fully qualified name includes the model and any category names in which this table exists:

Model	Category	Table	Fully qualified Name
CRM	NorthAmerica	Employees	`CRM.NorthAmerica.Employees`
CRM	NorthAmerica	Customers	`CRM.NorthAmerica.Customers`
VendorBooks	Book	Authors	`VendorBooks.Book.Authors`
VendorBooks	Book	Publishers	`VendorBooks.Book.Publishers`

You can use unqualified table metadata identifiers only if the partial name is unambiguous in the context in which you use it.

Model	Category	Table	Fully Qualified Name
CRM	NorthAmerica	Employees	Employees

Metadata Columns

You can also specify columns by fully qualified names in your queries. As with tables, these fully qualified names include the table metadata identifier, including model, categories, and table names, to which this column belongs.

Metadata Table Identifier			Column	Fully Qualified Name
CRM	NorthAmerica	Employees	EmployeeID	`CRM.NorthAmerica.Employees.EmployeeID`
CRM	NorthAmerica	Employees	FirstName	`CRM.NorthAmerica.Employees.FirstName`
VendorBooks	Book	Authors	Author_ID	`VendorBooks.Book.Authors.Author_ID`

You can use the unqualified column name, that is, only the last part of the fully qualified column metadata identifier, where that name is unambiguous.

If you use the shorter name, however, the column name must exist in exactly one table used in the query's FROM clause.

Metadata Table Identifier			Column	Unqualified Name
CRM	NorthAmerica	Employees	EmployeeID	`EmployeeID`

Aliases as Metadata Identifiers

If you use the SQL AS keyword to create an alias for a table identifier, that alias acts as the table metadata identifier, and the alias acts as a fullyqualified table metadata identifier.

Metadata Table Identifier			Column	Fully Qualified Name
CRM	NorthAmerica	Employees	NAEmployees	`NAEmployees`

You can use this alias name in place of a table metadata identifier in a fully qualified element metadata identifier:

```
NAEmployees.EmployeeID
```

Quotation Marks within Identifiers

You can enclose your metadata identifiers, such as default system alias metadata identifier, fully qualified table metadata identifier, or fully qualified column identifier, within quotation marks:

```
“CRM.NorthAmerica.Employees”.”LastName”
```

```
"VendorBooks"."VendorBooks.Authors"
```

```
VendorBooks."VendorBooks.Publishers"
```

You can include alias names in quotation marks:

```
“NAEmployees”.EmployeeID
```

You cannot, however, use quotation marks around unqualified table metadata identifiers or column metadata identifiers because these are indistinguishable from a string literal in a query:

```
“Employees”
```

```
“LastName”
```

Prohibited Characters in Identifiers

Blank Spaces

You cannot include blank spaces in identifiers, even if you enclose them in quotation marks. For example, these are not valid identifiers:

```
CRM.NorthAmerica.Employees.Last Name
```

```
“CRM.NorthAmerica.Employees”.”Last Name”
```

Brackets

You cannot enclose a part of the fully qualified identifier in brackets. For example, this is not a valid identifier:

```
CRM.NorthAmerica.Employees.[LastName]
```

JOINS

Explicit Joins

By default, a query uses explicit joins between all tables specified in the FROM clause. The FROM clause can specify tables that do not have join conditions or any other criteria. In this case, the DQE joins the tables by cross product. Otherwise, the DQE joins tables by the explicit criteria. In addition, you can specify the join type in the FROM clause to use INNER, RIGHT OUTER, LEFT OUTER, or FULL OUTER joins. These semantics match standard SQL definitions.

Join Processing

The SAP Distributed Query Engine pushes joins down to back-end systems that support them whenever possible. The metadata model for each system captures whether that system supports joins. If a query to the DQE requests multiple tables from a single system, the join occurs in the system, not in the DQE. For example, take this query:

```
SELECT SystemA.Table1.ColA, SystemA.Table2.ColA
```

```
FROM SystemA.Table1, SystemA.Table2
```

```
WHERE SystemA.Table1.ColA = SystemA.Table2.ColA
```

The query uses two tables (Table1 and Table2) from a single sysem. If SystemA is a relational database that supports joins, the joins occur entirely at the source. On the other hand, if SystemA is not a relational database or some other source that supports joins, then the DQE performs the join.

Dependent Join

You can use a special optimization called a dependent join to reduce the rows returned when you do not specify conditions on a table. When you join two tables and do not specify conditions on one, the DQE evaluates the first table’s query. It then uses the returned rows to form a query with an IN predicate (AKA a set criteria) to retrieve the rows from the second query. In most cases, this makes the second query much more specific and will return many fewer rows, reducing the communication and join processing costs. However, if the number of rows returned from the first query exceeds the second data source’s Max Set Criteria Size as defined in the metadata model, the DQE throws an exception during query processing.

Implicit Type Conversions During Join

If joins are specified between two columns of different types and a known type conversion exists, the join will proceed as specified and the column will automatically be converted. For instance, if a join is made between Column1 of type Integer and Column2 of type Long, the data from Column 1 will be converted to type Long to perform the join.

EXPRESSIONS

You can use expressions almost anywhere in a query, including the following clauses:

- SELECT
- FROM (if specifying join criteria)
- WHERE
- HAVING

You cannot, however, use expressions in the ORDER BY clause.

An expression is either a constant, a function, or an column.

A function usually comprises a function name and a list of arguments, which

are expressions. You can nest expressions using functions. For example:

```
SELECT 5 + c1 FROM model.table
SELECT CONCAT(fname, CONCAT(' ', lname)) FROM HR.Employees
SELECT c1 FROM m.g1 JOIN n.g2 ON 5+m.g1.j=n.g2.j
SELECT * FROM model.table WHERE date > NOW()
SELECT 5 + CONVERT(c1, integer) FROM m.g1
```

DATATYPES

DQE runtime datatypes include the following:

Type	Value
string	String of unicode characters of arbitrary length.
Char	A single bit, 2-byte character
Boolean	A single bit, or boolean, with two possible values.
byte	Numeric, integral type, signed 8-bit.
Short	Numeric, integral type, signed 16-bit.
Integer	Numeric, integral type, signed 32-bit.
long	Numeric, integral type, signed 64-bit.
biginteger	Numeric, integral type, arbitrary precision.
Float	Numeric, floating point type, 32-bit IEEE 754 floating-point numbers.
Double	Numeric, floating point type, 64-bit IEEE 754 floating-point numbers.
Bigdecimal	Numeric, floating point type, arbitrary precision.
date	Datetime, representing a single day (year, month, day).

Type	Value
time	Datetime, representing a single time (hours, minutes, seconds, milliseconds).
Timestamp	Datetime, representing a single date and time (year, month, day, hours, minutes, seconds,milliseconds, nanoseconds).
object	Any arbitrary Java object, must implement java.lang.Serializable.

DATATYPE CONVERSIONS

The SAP Distributed Query Engine supports certain implicit datatype conversions in which your query does not need to use explicit function-based conversions.

The following table describes the implicit datatype conversions supported, the explicit types you can perform using functions, and datatypes conversions not allowed at all:

Source Datatype \ Target Datatype	string	char	boolean	byte	short	integer	long	biginteger	float	double	bigdecimal	date	time	timestamp	object
string		C	C	C	C	C	C	C	C	C	C	C	C	C	N
char	I		N	N	N	N	N	N	N	N	N	N	N	N	N
boolean	I	N		C	C	C	C	C	C	C	C	N	N	N	N
byte	I	N	C		I	I	I	I	I	I	I	N	N	N	N
short	I	N	C	C		I	I	I	I	I	I	N	N	N	N
integer	I	N	C	C	C		I	I	I	I	I	N	N	N	N
long	I	N	C	C	C	C		I	I	I	I	N	N	N	N
biginteger	I	N	C	C	C	C	C		I	I	I	N	N	N	N
float	I	N	C	C	C	C	C	C		I	I	N	N	N	N
double	I	N	C	C	C	C	C	C	C		I	N	N	N	N
bigdecimal	I	N	C	C	C	C	C	C	C	C		N	N	N	N
date	I	N	N	N	N	N	N	N	N	N	N		N	C	N
time	I	N	N	N	N	N	N	N	N	N	N	N		C	N
timestamp	I	N	N	N	N	N	N	N	N	N	N	C	C		N
object	N	N	N	N	N	N	N	N	N	N	N	N	N	N	

I indicates you can convert the types either implicitly or with keyword.

C indicates you can convert the types only with a CAST or CONVERT statement.

N indicates you cannot convert the types.

Implicit Conversion

The SAP Distributed Query Engine automatically adds casting logic for implicit conversions. You can, however, use the CAST or CONVERT functions at your discretion in implicit conversions to explicitly use the function you want. For more information about these functions, see the section “Functions.”

Note that some implicit conversions, such as from a string to a date, require a specific string literal to work. If the string differs from the required literal format, you can often find a function to perform the conversion, such as parseDate.

For more information about the specific literals required for implicit conversion, see “Special Terminals.”

Conversion of String Literals

The DQE automatically converts string literals within a SQL statement to their implied types. This typically occurs in a criteria comparison where an column with a different datatype is compared to a literal string:

```
SELECT * FROM my.table WHERE created_by = '2003-01-02'
```

For example, if the created_by column has the datatype of date, the DQE automatically converts the string literal to a date datatype as well.

Note that some literal conversions require a specific string literal to work.

For more information about the specific literals required for implicit conversion, see “Special Terminals.”

Precision Loss in Conversion

When converting the types, you can keep in mind this general rule for DQE conversions: the DQE allows implicit conversion when no precision is lost, but requires explicit casting when the conversion might cause a loss of precision.

Casting Float and Integers to Boolean

You can cast floating-point numeric datatypes (float) and integers to and from boolean datatypes. When you do so, the DQE converts these datatypes as follows:

float or integer	**boolean**
0	false
1	true
other	error

Casting Literal Strings to Boolean

The DQE can also automatically convert literal strings to boolean values as follows:

Literal String	**boolean value**
‘true’	true
‘false’	false

Handling Date Datatypes

Implicit Conversion from String

The DQE only implicitly converts literal strings formatted as follows to the associated date-related datatypes:

Literal String Formats	**Implicit Conversion To**
“yyyy-mm-dd”	DATE
“hh:mm:ss”	TIME
“yyyy-mm-dd hh:mm:ss.SSSSSSSSS”	TIMESTAMP

Explicit Definitions of Date Datatypes

Your query can also explicitly declare a datatype for a string:

Datatype	Explicit Declaration
DATE	{d'2002-01-23'}
TIME	{t'11:42:01'}
TIMESTAMP	{ts'2002-01-21 11:42:01.5'}

Parsing Date Datatypes from Strings

The DQE does not implicitly convert strings that contain dates presented in different formats, such as '19970101' and '31/1/1996' to date-related datatypes. You can, however, use the parseDate, parseTime, and parseTimestamp functions, described in the next section, to explicitly convert strings with a different format to the appropriate datatype.

These functions use the convention established within the java.text.SimpleDateFormat class to define the formats you can use with these functions. You can learn more about how this class defines date and time string formats by visiting the Sun Java Web site at http://java.sun.com/j2se/1.3/docs/api/java/text/SimpleDateFormat.html.

For example, you could use these function calls, with the formatting string that adheres to the java.text.SimpleDateFormat convention, to parse to parse strings and return the datatype you need:

String	Function Call to Parse String
“19970101”	parseDate(myDate, “yyyyMMdd”);
“31/1/1996”	parseDate(myDate, “dd’/’MM’/’yyyy”);
“22:08:56 CST”	parseTime (myTime, “HH:mm:ss z”);
“03.24.2003 at 06:14:32”	parseTimestamp(myTimestamp, "MM.dd.yyyy 'at' hh:mm:ss");

For more information about these functions, see “Date/Time” in the section “Functions.”

Parsing Numeric Datatypes from Strings

The DQE offers a set of functions you can use to parse numbers from strings and deliver a numeric datatype containing that number.

For each string, you need to provide the formatting of the string so that the DQE knows how to read the string to find the number within it.

These functions use the convention established within the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at http://java.sun.com/j2se/1.3/docs/api/java/text/DecimalFormat.html.

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to parse to parse strings and return the datatype you need:

Input String	Function Call to Format String	Output Value	Output Datatype
“$25.30”	parseDouble(cost, “$#,##0.00;($#,##0.00)”)	25.3	double
“25%”	parseFloat(percent, “#,##0%”)	25	float
“2,534.1”	parseFloat(total, “#,##0.###;-	2534.1	float

Input String	Function Call to Format String	Output Value	Output Datatype
	#,##0.###")		
"1.234E3"	parseLong(amt, "0.###E0")	1234	long
"1,234,567"	parseInteger(total, "#,##0;-#,##0")	1234567	integer

Formatting Numeric Datatypes as Strings

The DQE offers a set of functions you can use to convert numeric datatypes into strings containing that number. For each string, you need to provide the formatting so that the DQE knows how to output the string in the form you want.

These functions use the convention established within the java.text.DecimalFormat class to define the formats you can use with these functions. You can learn more about how this class defines numeric string formats by visiting the Sun Java Web site at http://java.sun.com/j2se/1.3/docs/api/java/text/DecimalFormat.html.

For example, you could use these function calls, with the formatting string that adheres to the java.text.DecimalFormat convention, to format the numeric datatypes into strings:

Input Value	Input Datatype	Function Call to Format String	Output String
25.3	float	formatFloat(cost, "$#,##0.00;($#,##0.00)")	"$25.30"
0.25	double	formatDouble(percent, "#,##0%")	"25%"
2534.1	double	formatDouble(total, "#,##0.###;-#,##0.###")	"2,534.1"
1234	float	formatFloat(amt, "0.###E0")	"1.234E3"
1234567	integer	formatInteger(total, "#,##0;-#,##0")	"1,234,567"

FUNCTIONS

The categories of standard DQE functions include:

- Numeric
- String
- Date/Time
- Conversion
- Miscellaneous

Numeric

Function	Definition	Datatype Constraint
`+ - * /`	Standard numeric operators	x in {integer, long, float, double, biginteger,bigdecimal}, return type is same as x
`ABS(x)`	Absolute value of x. x in {integer, long, float, double, biginteger, bigdecimal},	return type is same as x
`ACOS(x)`	Returns the arccosine of x. x in {double},	return type is double
`ASIN(x)`	Returns the arcsine of x x in	return type is double

Function	Definition	Datatype Constraint
	{double},	
ATAN(x)	Returns the arctangent of x x in {double},	return type is double
ATAN2(x, y)	Returns the arctangent of x/y x, y in {double},	return type is double
CEILING(x)	Ceiling of x. x in {double, float},	return type is double
COS(x)	Returns the cosine of x x in {double},	return type is double
COT(x)	Returns the cotangent of x x in {double},	return type is double
DEGREES(x)	Converts x radians to degrees x in {double},	return type is double
EXP(x)	e^x. x in {double, float},	return type is double
FLOOR(x)	Floor of x. x in {double, float},	return type is double
FORMATBIGDECIMAL(x, y)	Formats x using format y. x is bigdecimal, y is string,	returns string
FORMATBIGINTEGER(x, y)	Formats x using format y. x is biginteger, y is string,	returns string
FORMATDOUBLE(x, y)	Formats x using format y. x is double, y is string,	returns string
FORMATFLOAT(x, y)	Formats x using format y.x is float, y is string,	returns string
FORMATINTEGER(x, y)	Formats x using format y.	x is integer, y is string, returns string
FORMATLONG(x, y)	Formats x using format y.x is long, y is string,	returns string
LOG(x)	Natural log of x (base e).x in {double, float},	return type is double
LOG10(x)	Log of x (base 10). x in {double, float},	return type is double
MOD(x, y)	Modulus (remainder of x / y). x in {integer, long, float, double, biginteger},	return type is same as x
PARSEBIGDECIMAL(x, y)	Parses x using format y. x, y are strings,	returns bigdecimal
PARSEBIGINTEGER(x, y)	Parses x using format y. x, y are strings,	returns biginteger
PARSEDOUBLE(x, y)	Parses x using format y. x, y are strings,	returns double
PARSEFLOAT(x, y)	Parses x using format y. x, y are strings,	returns float

Function	Definition	Datatype Constraint
PARSEINTEGER(x, y)	Parses x using format y. x, y are strings,	returns integer
PARSELONG(x, y)	Parses x using format y. x, y are strings,	returns long
PI()	Returns value of pi	return type is double
POWER(x, y)	x to the y power. x in {integer, long, float, double, biginteger},	if x is biginteger then return type is big integer, else double
RADIANS(x)	Converts x degrees to radiansx in {double},	return type is double
SIGN(x)	1 if x > 0, 0 if x = 0-1 if x < 0. x in {integer, long, float, double, biginteger, bigdecimal},	return type is integer
SIN(x)	Returns the sine of x x in {double},	return type is double
SQRT(x)	Square root of x. x in {double, float},	return type is double
TAN(x)	Returns the tangent of x x in {double},	return type is double

String

Unless specified, all of the arguments and return types in the following table are strings and all indexes are 0-based.

Function	Definition	Datatype Constraint
x || y	Concatenation operator	x,y in {string}, return type is string
ASCII(x)	Provides ASCII value of character x	return type is integer
CHR(x) CHAR(x)	Provides the character for ASCII value x	x in {integer}
CONCAT(x, y)	Concatenation of x and y	
INITCAP(x)	Make first letter of each word in string x capital and all others lowercase	
INSERT(string1, start, length, string2)	Deletes length characters from string1 at start and inserts string2	start, length in {integer}
LCASE(x) LOWER(x)	Lowercase of x	
LEFT(x, y)	Get left y characters of x	
LENGTH(x)	Length of x	return type is integer
LOCATE(x, y)	Find position of x in y starting at beginning of y	

Function	Definition	Datatype Constraint
LOCATE(x, y, z)	Find position of x in y starting at z	z in {integer}
LPAD(x,y)	Pad string x with spaces on the left to the size of y	y in {integer}
LPAD(x,y,z)	Pad string x with character z on the left to the size of y	y in {integer}
LTRIM(x)	Left trim x of white space	
REPEAT(x, y)	Return x, repeated y times	y in {integer}
REPLACE(x, y, z)	Replace all y in x with z	
RIGHT(x, y)	Get right y characters of x. y in {integer}	
RPAD(x,y)	Pad string x with spaces on the right tothe size of y.	y in {integer}
RPAD(x,y,z)	Pad string x with character z on the rightto the size of y	y in {integer}
RTRIM(x)	Right trim x of white space	
SPACE(x)	Return string of x spaces	x in {integer}
SUBSTRING(x, y)	Get substring from x, from position y to the end of x	y in {integer}
SUBSTRING(x, y, z)	Get substring from x from position y with length z	y, z in {integer}
TRANSLATE(x,y, z)	Translate string x by replacing each character in y with the character in z at the same position.	
UCASE(x) UPPER(x)	Uppercase of x	

Date/Time

Function	Definition	Datatype Constraint
CURDATE()	Returns current date	returns date
CURTIME()	Returns current time	returns time
NOW()	Returns current timestamp (date and time)	returns timestamp
DAYNAME(x)	Returns name of day (such as Monday)	x in {date, timestamp}, returns string
DAYOFMONTH(x)	Returns day of month (such as 30). 1-based	x in {date, timestamp}, returns integer
DAYOFWEEK(x)	Returns day of week (such as 1 for Sunday). 1-based	x in {date, timestamp}, returns integer
DAYOFYEAR(x)	Returns day of year (such as 200).	x in {date, timestamp}, returns

Function	Definition	Datatype Constraint
	1-based	integer
FORMATDATE(x, y)	Formats date x using format y	x is date, y is string, returns string
FORMATTIME(x, y)	Formats time x using format y	x is time, y is string, returns string
FORMATTIMESTAMP(x, y)	Formats timestamp x using format y	x is timestamp, y is string, returns string
HOUR(x)	Returns hour (such as 13 for 1:00 pm). 0-based	x in {time, timestamp}, returns integer
MINUTE(x)	Returns minute (such as 15). 0-based	x in {time, timestamp}, returns integer
MONTH(x)	Returns month (such as 1 for January 22nd). 1-based	x in {date, timestamp}, returns integer
MONTHNAME(x)	Returns name of month (such as January)	x in {date, timestamp}, returns string
PARSEDATE(x, y)	Parses date from x using format y	x, y in {string}, returns date
PARSETIME(x, y)	Parses time from x using format y	x, y in {string}, returns time
PARSETIMESTAMP(x, y)	Parses timestamp from x using format y	x, y in {string}, returns timestamp
QUARTER(x)	Returns quarter of the year x is in	x in {date, timestamp}, returns type is integer range 1-4
SECOND(x)	Returns seconds (such as 30). 0-based	x in {time, timestamp}, returns integer
TIMESTAMPADD(x, y, z)	Start with z, add y intervals of type x. x is code with following definition: **SQL_TSI_FRAC_SECOND** - fractional seconds (billionths of a second) **SQL_TSI_SECOND** – seconds **SQL_TSI_MINUTE** – minutes **SQL_TSI_HOUR** – hours **SQL_TSI_DAY** – days **SQL_TSI_WEEK** – weeks **SQL_TSI_MONTH** – months **SQL_TSI_QUARTER** - quarters (3 months) **SQL_TSI_YEAR** - years The constant x may be specified either as a string literal or a constant value.	z in {date, time, timestamp} x in {string}, y in {integer}, return type matches type of z
TIMESTAMPDIFF(x, y, z)	Return approximate number of intervals of type x between y and z.	x in {integer}, y, z in {date, time, timestamp}, return type is a long

Function	Definition	Datatype Constraint
	x is code as specified in TIMESTAMPADD function. The constant x may be specified either as a string literal or a constant value. The value returned is approximate and will be less accurate over longer time periods.	
`WEEK(x)`	Returns week in year (such as 1 for 1st week). 1-based	x in {date, timestamp}, returns integer
`YEAR(x)`	Returns year (such as 2001)	x in {date, timestamp}, returns integer

Type

Within your queries, you can convert between datatypes using the CONVERT or CAST keyword. These keywords have the following syntax:

Function	Definition	Datatype Constraint
`CONVERT(x, type)`	Converts x to type, where type is a DQE runtime datatype	
`CAST(x AS type)`	Converts x to type, where type is a DQE runtime datatype	

These functions are identical other than syntax; CAST is the standard SQL syntax, CONVERT is the standard JDBC/ODBC syntax.

Miscellaneous

Decode Functions

The Decode functions allow you to have the DQE examine the contents of a column in a result set and alter, or decode, the value so that your application can better use the results.

Function	Definition	Datatype Constraint
`DECODESTRING(x, y)`	Decode column x using string of value pairs y and return the decoded column as a string	
`DECODESTRING(x, y, z)`	Decode column x using string of value pairs y with delimiter z and return the decoded column as a string	
`DECODEINTEGER(x, y)`	Decode column x using string of value pairs y and return the decoded column as an integer	
`DECODEINTEGER(x,y,z)`	Decode column x using string of value pairs y with delimiter z and return the decoded column as an integer	

Within each function call, you include the following arguments:

- x is the input value for the decode operation. This will generally be a column name.
- y is the literal string that contains a delimited set of input values and output values.
- z is an optional parameter on these methods that allows you to specify what delimiter the string specified in y uses.

For example, your application might query a table called `PARTS` that contains a column called `IS_IN_STOCK` which contains a boolean value that you need to change into an integer for your application to process. In this case, you can use the `DECODEINTEGER` function to change the boolean values to integers:

```
SELECT DECODEINTEGER(IS_IN_STOCK, 'false, 0, true, 1') FROM
PartsSupplier.PARTS;
```

When the DQE encounters the value `false` in the result set, it replaces the value with 0.

If, instead of using integers, your application requires string values, you can use the `DECODESTRING` function to return the string values you need:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false, no, true, yes, null') FROM
PartsSupplier.PARTS;
```

This sample query provides, in addition to two input/output value pairs, a value to use if the column does not contain any of the preceding input values.

If the row in the `IS_IN_STOCK` column does not contain `false` or `true`, the SAP Distributed Query Engine inserts a null into the result set.

When you use these decode functions, you can provide as many input/output value pairs if you want within the string. By default, the DQE expects a comma delimiter, but you can add a third parameter to the function call to specify a different delimiter:

```
SELECT DECODESTRING(IS_IN_STOCK, 'false:no:true:yes:null',':') FROM
PartsSupplier.PARTS;
```

You can use keyword **`null`** in the decode string as either an input value or an output value to represent a null value. However, if you need to use the literal string "null" as an input or output value, which means the word null appears in the column and not a null value, you can put the word in quotes: "null".

```
SELECT DECODESTRING(IS_IN_STOCK,
'null,no,"null",no,nil,no,false,no,true,yes') FROM PartsSupplier.PARTS;
```

If the decode function does not find a matching output value in the column and you have not specified a default value, the decode function will return the original value that the DQE found in that column.

Determining Null Values

These functions help you determine if an attribute or variable contains a null value:

Function	Definition	Datatype Constraint
`NVL(x,y)` `IFNULL(x,y)`	If x is null, return y; else return x	x, y, and the return type must be the same type but can be any type

Query Grammar

Notation

The following table defines symbols used in the grammar description. A valid sentence in the grammar expands <query> that is all terminals.

Notation	Definition
xxxx	a non-terminal
XXXX	a terminal string
XXXX	a special terminal (see definitions below)
[ ]	an optional clause (0 or 1 time)
( )	a grouping clause
|	a logical or
*	a repetition of the previous clause 0 or more times
+	a repetition of the previous clause 1 or more times

Terminals are symbols in the grammar that cannot be expanded, such as literal values or reserved words. A valid query must contain only terminals.

Most terminals are just reserved word strings, like “SELECT”, or punctuation, like “,”. Special terminals are just terminals that require some explanation.

Special Terminals

Terminal	Definition
ID	A dotted metadata identifier, such as Table.Column, <Default_System_Alias>.Table.Column, Table, <Default_System_Alias>.Table. For more information about metadata identifiers, see the section “Metadata Identifiers.”
TABLEID	A dotted metadata ID ending in *. This special ID represents all the columns in the specified table, such as Table.* or <Default_System_Alias>.Table.*.
STRING	A string value in ‘ ‘ or “ “, such as ’string x’ or “string y”
INTEGER	An integer value, such as 157
FLOAT	A real value, such as 157.32 or –12.6e5
DATE	A date literal: {d ‘2002-01-23’}
TIME	A time literal: {t ‘11:42:01’}
TIMESTAMP	A timestamp literal: {ts ‘2002-01-21 11:42:01.5’}
BOOLEAN	A boolean literal {b ‘true’}
FUNCTIONNAME	A user-defined function name

Non-Terminals

Non-terminal symbols in the grammar that are expanded into terminals and non-terminals.

The DQE supports the following non-terminal symbols:

Non-Terminal	Definition
`command`	query | execute
`query`	setQuery [orderby] [option]
`setQuery`	queryStatement | queryStatement UNION [ALL] queryStatement | (queryStatement)
`queryStatement`	setQuery | selectStatement
`selectStatement`	select from [where] [groupby] [having]
`select`	SELECT [DISTINCT] (* | (selectClause (, selectClause)*))
`selectClause`	TABLEID | idAlias | expressionAlias | aggregateAlias
`from`	FROM fromClause (, fromClause)*
`fromClause`	idAlias | joinClause | subqueryClause
`joinClause`	fromClause joinType fromClause ON joinCrit | fromClause [CROSS] JOIN fromClause
`subqueryClause`	(query | execute) AS ID
`joinType`	([INNER] | ((RIGHT | LEFT | FULL) [OUTER])) JOIN
`where`	WHERE logicalCrit
`joinCrit`	compareCrit | compoundJoinCrit | (joinCrit)
`logicalCrit`	compareCrit | matchCrit | setCrit | isNullCrit | notCrit | compoundCrit | (logicalCrit)
`compareCrit`	expression (= | <> | < | <= | > | >) expression
`matchCrit`	expression LIKE expression [ESCAPE STRING]
`isNullCrit`	expression IS NULL
`setCrit`	expression IN (expression (, expression)*)
`subquerySetCrit`	expression IN (query | execute)
`notCrit`	NOT (logicalCrit)
`compoundCrit`	compoundCritAnd | compoundCritOr
`compoundCritAnd`	logicalCrit AND logicalCrit (AND logicalCrit)*
`compoundCritOr`	logicalCrit OR logicalCrit (OR logicalCrit)*
`compoundJoinCrit`	compareCrit AND compareCrit (AND compareCrit)*
`orderby`	ORDER BY ID [ASC | DESC](, ID [ASC | DESC])*

Non-Terminal	Definition
groupby	GROUP BY ID (, ID)*
having	HAVING logicalCrit
option	OPTION [SHOWPLAN] [MAKEDEP ID (, ID)*]
execute	(EXEC | EXECUTE) ID ([param (, param)*])
literal	STRING | INTEGER | FLOAT |DATE | TIME | TIMESTAMP | BOOLEAN
idAlias	ID [AS ID]
expressionAlias	expression [AS ID]
aggregateAlias	aggregateFunction [AS ID]
expression	literal | ID | scalarFunction
scalarFunction	function | cast | expression + expression1 expression - expression1 expression * expression1 expression / expression1 expression || expression1
function	FUNCTIONNAME ([expression (,expression)*])
cast	CAST (expression AS type)
type	string | boolean | byte | short | char | integer | long | biginteger | float | decimal | bigdecimal | date | time | timestamp | object