

Oracle® Fusion Middleware

Logical SQL Reference Guide for Oracle Business Intelligence Enterprise Edition

Release 12c

E71870-03

August 2016

The Logical SQL Reference Guide provides syntax and usage information for the Logical SQL statements understood by the Oracle BI Server. Logical SQL includes standard SQL, plus special functions (SQL extensions) such as `AGO`, `TODATE`, `EVALUATE`, and others. Logical SQL queries resolve to Presentation layer objects.

This guide contains the following topics:

- [About Logical SQL in Oracle Business Intelligence](#)
- [SQL Syntax and Semantics](#)
- [Aggregate, Running Aggregate, Time Series, and Reporting Functions](#)
- [String Functions](#)
- [Math Functions](#)
- [Calendar Date/Time Functions](#)
- [Conversion Functions](#)
- [Lookup Functions](#)
- [Database Functions](#)
- [Hierarchy Navigation Functions](#)
- [System Functions](#)
- [Advanced Analytics Internal Logical SQL Functions](#)
- [Advanced Analytics External Logical SQL Functions](#)

About Logical SQL in Oracle Business Intelligence

Provides an overview for defining SQL Select statements to create expressions for derived columns.

The Oracle BI Server accepts SQL `SELECT` statements from client tools. Additionally, the Oracle BI Administration Tool enables you to define logical columns with complex expressions. This guide explains the syntax and semantics for the `SELECT` statement and for the expressions you can use in the Administration Tool to create derived columns.

The abstraction provided by the Presentation layer and Business Model and Mapping layer enables clients to query data with Logical SQL only, so that the interaction with actual physical sources is handled by the Oracle BI Server. The complexity of the multiple source languages needed to communicate with each data source type is hidden from users and clients.

In Administration Tool, you can view the Logical SQL queries issued by Oracle BI Server for particular analyses by viewing the SQL Issued section of the Advanced tab of the Analysis editor. If you have the appropriate privileges, then you can also view SQL by displaying the Manage Sessions page in the Administration tab. Click **View Log** from the Manage Sessions page to see further details.

In Answers , there are also several places where you can issue Logical SQL. If you have the appropriate privileges, then you can use the Issue SQL page in the Administration tab to enter any SQL code to send to the Oracle BI Server. If an analysis does not contain hierarchical columns, member selections, or groups, then you can use the Advanced SQL Clauses fields in the Advanced tab of the Analysis editor. You can also enter SQL in the New Filter dialog.

In the Administration Tool, Logical SQL appears mostly in the form of expressions related to objects in the Business Model and Mapping layer. You create SQL functions in Expression Builder.

Other clients, like Oracle BI Publisher, Oracle's Hyperion Interactive Reporting, the Oracle BI Add-in for Microsoft Office, and Essbase, also provide their own interfaces to view and issue Logical SQL to the Oracle BI Server.

SQL Syntax and Semantics

These sections explain SQL syntax and semantics.

The following topics are included:

- [Syntax and Usage Notes for the SELECT Statement](#)
- [Syntax and Usage Notes for SELECT_PHYSICAL](#)
- [Limiting and Offsetting Rows Returned](#)
- [Rules for Queries with Aggregate Functions](#)
- [Operators](#)
- [Conditional Expressions](#)
- [Expressing Literals](#)
- [Calculated Members](#)
- [Variables](#)

Syntax and Usage Notes for the SELECT Statement

The SELECT statement or query specification is the way to query a decision support system through the Oracle BI Server.

A SELECT statement returns a table to the client that matches the query. It is a table in the sense that the results are in the form of rows and columns.

The `SELECT` statement is the basis for querying any structured query language (SQL) database. The Oracle BI Server accepts logical requests to query objects in a repository, and users or query tools make those logical requests with ordinary SQL `SELECT` statements. The server then translates the logical requests into physical queries against one or more data sources, combines the results to match the logical request, and returns the answer to the end user.

The `SELECT` statement in Logical SQL differs from standard SQL in that it is not necessary to join the tables to return results. Any join conditions supplied in the query are ignored because the join conditions are predefined in the .

This section provides the basic syntax for the `SELECT` statement, as well as definitions for individual clauses. The syntax descriptions cover only basic syntax and features unique to the Oracle BI Server. For a more comprehensive description of SQL syntax, see a third-party reference book on SQL or a reference manual on SQL from your database vendors. For Oracle Database, see the Oracle Database SQL Language Reference.

This section contains the following topics:

- [Basic Syntax for the SELECT Statement](#)
- [Usage Notes](#)
- [Subquery Support](#)
- [SELECT List Syntax](#)
- [FROM Clause Syntax](#)
- [WHERE Clause Syntax](#)
- [GROUP BY Clause Syntax](#)
- [ORDER BY Clause Syntax](#)

Basic Syntax for the SELECT Statement

Use this syntax to write a basic Select statement.

Syntax for the `SELECT` statement is as follows:

```
SELECT [DISTINCT] select_list
FROM from_clause
[WHERE search_condition]
[GROUP BY column {, column}
  [HAVING search_condition]]
[ORDER BY column {, column}]
```

Where:

select_list is the list of columns specified in the request. See [SELECT List Syntax](#) for more information.

FROM *from_clause* is the list of tables in the request. Optionally includes certain join information for the request. See [FROM Clause Syntax](#) for more information.

WHERE *search_condition* specifies any combination of conditions to form a conditional test. A WHERE clause acts as a filter that lets you constrain a request to obtain results that answer a particular question. Together with the columns you select,

filters determine the contents of your results. See [WHERE Clause Syntax](#) for more information.

`GROUP BY column {, column}` specifies a column or alias belonging to a table defined in the data source. See [GROUP BY Clause Syntax](#) for more information.

`HAVING search_condition` specifies any combination of conditions to form a conditional test. The syntax is identical to that for the `WHERE` clause.

`ORDER BY column {, column}` specifies the columns to order the results by. See [ORDER BY Clause Syntax](#) for more information.

Usage Notes

The Oracle BI Server treats the `SELECT` statement as a logical request.

If aggregated data is requested in the `SELECT` statement, a `GROUP BY` clause is automatically assumed by the server. Any join conditions supplied in the query are ignored because the join conditions are all predefined in the Oracle BI Repository.

The Oracle BI Server accepts the following SQL syntaxes for comments:

- `/**/` C-style comments
- `//` Double slash for single-line comments
- `#` Number sign for single-line comments

Subquery Support

The Oracle BI Server supports subqueries.

The Oracle BI Server supports certain subqueries, as well as `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT` operations in logical requests. Subquery support increases the range of business questions that you can answer, eases the formulation of queries, and provides some ability to query across multiple business models.

The Oracle BI Server supports the following subquery predicates in any conditional expression, for example, within `WHERE`, `HAVING`, or `CASE` statements:

```
IN, NOT IN
Any, >=Any, =Any, <Any, <=Any, <>Any
All, >=All, =All, <All, <=All, <>All
EXISTS, NOT EXISTS
```

In Answers, advanced users and developers can use the Advanced SQL Clauses fields in the Advanced tab of the Analysis editor to specify various SQL clauses such as `GROUP BY`, `HAVING`, and `DISTINCT`, to include in the SQL queries that are sent to the Oracle BI Server. If an analysis contains hierarchical columns, selections, or groups, then certain Advanced SQL Clauses fields are not available.

SELECT List Syntax

The `select_list` function lists the columns in the request.

You should select all columns from a single business model. You can include table names, for example, `Table.Column`. Table names are optional except when the column names are not unique in the business model. If column names contain spaces, enclose column names in double quotes. You do not need to include the `DISTINCT` keyword because the Oracle BI Server always performs a distinct query. You do not need to include the aggregation function such as `SUM`, for columns that are aggregated because

the Oracle BI Server knows the aggregation rules and aggregation is performed automatically.

Syntax

```
...
* |
  (column | expr) [[AS] alias]
  {, (column | expr) [[AS] alias] }
...
```

Where:

* Indicates all columns in the resultant table in the FROM clause.

column is a column (or alias) belonging to a table defined in the data source.

expr is any valid SQL expression.

Note:

You cannot use * to select all columns from the Advanced tab of the Analysis editor in Answers. Instead, you must specify particular columns.

FROM Clause Syntax

The Oracle BI Server accepts any valid SQL FROM clause syntax.

To simplify FROM clause creation, you can specify the name of a subject area instead of a list of tables. The Oracle BI Server determines the proper tables and the proper join specifications based on the columns the request asks for and the configuration of the Oracle BI Repository.

WHERE Clause Syntax

The Oracle BI Server accepts any valid SQL WHERE clause syntax.

There is no need to specify any join conditions in the WHERE clause, because the joins are all configured within the Oracle Business Intelligence repository. Any join conditions specified in the WHERE clause are ignored.

Oracle BI EE also supports the following subquery predicates in any conditional expression (WHERE, HAVING or CASE statements):

```
IN, NOT IN
Any, >=Any, =Any, <Any, <=Any, <>Any
All, >=All, =All, <All, <=All, <>All
EXISTS, NOT EXISTS
```

GROUP BY Clause Syntax

With auto aggregation on the Oracle BI Server, there is no need to submit a GROUP BY clause.

When no GROUP BY clause is specified, the GROUP BY specification defaults to all of the non-aggregation columns in the SELECT list. If you explicitly use aggregation functions in the select list, you can specify a GROUP BY clause with different columns and the Oracle BI Server computes the results based on the level specified in the GROUP BY clause.

For an in-depth explanation and some examples of using the `GROUP BY` clause in requests against the Oracle BI Server, see [Rules for Queries with Aggregate Functions](#).

ORDER BY Clause Syntax

The Oracle BI Server accepts any valid SQL `ORDER BY` clause syntax, including referencing columns by their order in the select list such as `ORDER BY 3, 1, 5`.

In the `ORDER BY` clause, you can use the following syntax to alter the sort order for nulls in the query:

```
ORDER BY col1 NULLS LAST, ORDER BY col2 NULLS FIRST
```

For logical columns with sort order columns assigned to them, you can use the `ORDER BY` clause to disregard the sort order column and instead sort by the column's value. Note the following syntax:

```
ORDER BY { { <column_index> | <expr> } [ DISPLAY | SORTKEY ] [ ASC | DESC ] [NULLS { FIRST | LAST } ] }
```

Where:

`DISPLAY` sorts based on the order of the display value of the expression regardless of whether a sort column is assigned to the logical column. By default, Oracle BI Server assumes `DISPLAY` when a sort column *is not set* for the logical column.

`SORTKEY` sorts based on the logical column's assigned sort column. By default, Oracle BI Server assumes `SORTKEY` when a sort column *is set* for the logical column.

Syntax and Usage Notes for SELECT_PHYSICAL

The Oracle BI Server performs parsing, interpretation, and query generation on a `SELECT_PHYSICAL` query before passing it to the database.

Though a `SELECT_PHYSICAL` query bypasses the Presentation layer and the Business Model and Mapping layer, the Oracle BI Server still performs parsing, interpretation, and query generation on a `SELECT_PHYSICAL` query before passing it to the database.

A `SELECT_PHYSICAL` command can contain any element allowed in standard Oracle BI Server SQL with the following constraints:

- The `SELECT_PHYSICAL` command does not explicitly reference structures in the repository Business Model and Mapping layer or the Presentation layer
- The `SELECT_PHYSICAL` command does not require implicit logical transformation
- The `SELECT_PHYSICAL` command cannot contain certain aggregate functions, see [Aggregate Functions Not Supported in SELECT_PHYSICAL Queries](#) for details

Note:

`SELECT_PHYSICAL` statements are not cached.

You can set up an ODBC connection to the Oracle BI Server as a dedicated physical connection over which all `SELECT` queries are treated as `SELECT_PHYSICAL` queries. To do this, select **Route Requests To Physical Layer** in the ODBC data source for the Oracle BI Server. See “Integrating Other Clients with Oracle Business Intelligence” in *Integrator’s Guide for Oracle Business Intelligence Enterprise Edition* for more information.

`SELECT_PHYSICAL` statements are logged as Physical Request entries.

The topics in this section are the following:

- [Syntax for the `SELECT_PHYSICAL` Statement](#)
- [Aggregate Functions Not Supported in `SELECT_PHYSICAL` Queries](#)
- [Queries Supported by `SELECT_PHYSICAL`](#)
- [Using the `NATURAL_JOIN` Keyword](#)
- [Special Usages of `SELECT_PHYSICAL`](#)

Syntax for the `SELECT_PHYSICAL` Statement

The `SELECT_PHYSICAL` function syntax is like to the syntax used with a `SELECT` statement.

Basic syntax for `SELECT_PHYSICAL` queries is equivalent to [Basic Syntax for the `SELECT` Statement](#) with the term `SELECT_PHYSICAL` replacing the word `SELECT`, namely:

```
SELECT_PHYSICAL [DISTINCT] select_list
FROM from_clause
[WHERE search_condition]
[GROUP BY column {, column}
  [HAVING search_condition]]
[ORDER BY column {, column}]
```

Note:

The `SELECT_PHYSICAL` statement is close to the standard ANSI SQL `SELECT` statement. For example, you cannot omit the `GROUP BY` clause nor, where relevant, the `HAVING` clause in a `SELECT_PHYSICAL` aggregate query.

In `SELECT_PHYSICAL` queries, you must fully qualify the table names in the `FROM` list. Each fully qualified table name must match a table name in the physical layer of the repository.

A fully qualified table name consists of up to four components, database name, catalog name, schema name, and table name. Each component is surrounded by double quotes (") with a period (.) separator between components. For example, "SQL_DB"."My_Catalog"."My_Schema"."Customers" for a SQL Server table, and "FoodMart"... "Sales" for a cube table.

Aggregate Functions Not Supported in `SELECT_PHYSICAL` Queries

Provides a list of aggregate functions that are not supported by `SELECT_PHYSICAL` queries.

- AGO
- BOTTOMN
- FILTER
- FIRST
- LAST
- RCOUNT
- RMAX
- RMIN
- RSUM
- TODATE
- TOPN

Queries Supported by SELECT_PHYSICAL

The Oracle BI Server supports the use of SELECT_PHYSICAL for some logical query types.

The supported logical query types are:

- **Standard Non-Aggregate Queries**

Standard non-aggregate SELECT_PHYSICAL commands follow the same rules as standard non-aggregate SELECT commands. They can also include scalar functions, such as String, Math, and Calendar Date/Time functions. For example:

```
SELECT_PHYSICAL productid, categoryid
FROM "My_DB"."My_Schema"."products"
WHERE categoryid > 5;
```

```
SELECT_PHYSICAL LEFT(productname,10)
FROM "My_DB"."My_Schema"."products"
WHERE productname is not null;
```

- **Queries with Aggregate Functions**

In general, all aggregate functions supported in SELECT queries are also supported in SELECT_PHYSICAL queries. See [Aggregate Functions Not Supported in SELECT_PHYSICAL Queries](#) for a list of the exceptions to this rule.

For aggregates supported in SELECT_PHYSICAL commands, each aggregate must have an explicitly specified aggregation level, using the GROUP BY clause or the BY clause. For example:

```
SELECT_PHYSICAL employeeid, SUM(quantity by)
FROM "My_DB"."My_Schema"."employees";
```

```
SELECT_PHYSICAL employeeid, SUM(quantity)
FROM "My_DB"."My_Schema"."employees"
GROUP BY employeeid
```



```
HAVING SUM(quantity) > 100;
```

- **Subqueries**

The Oracle BI Server supports the following types of query:

- Queries where both the parent query and the subquery use `SELECT_PHYSICAL`
- Parent query uses `SELECT` and subquery uses `SELECT_PHYSICAL`

Subqueries are supported on both filters and on projections embedded in a Case statement.

For example:

```
SELECT_PHYSICAL *
FROM "My_DB"."My_Schema"."products"
WHERE supplierid IN
  (SELECT_PHYSICAL supplierid
   FROM "My_DB"."My_Schema"."suppliers");
```

```
SELECT productid
FROM snowflakesales.product
WHERE categoryid IN
  (SELECT_PHYSICAL categoryid
   FROM "My_DB"."My_Schema"."categories");
```

```
SELECT
CASE WHEN b.categoryid IN
  (SELECT_PHYSICAL a.categoryid
   FROM "My_DB"."My_Schema"."products" a)
THEN b.categoryid END
FROM categories b;
```

- **Queries with Derived Tables**

Both `SELECT` and `SELECT_PHYSICAL` queries can have derived tables in their `FROM` clause. You can create derived tables using either `SELECT` or `SELECT_PHYSICAL`. For example:

```
SELECT_PHYSICAL COUNT(DISTINCT t.rto)
FROM
  (SELECT_PHYSICAL employeed AS id, reportsto AS rto
   FROM "My_DB"."My_Schema"."employees") t;
```

```
SELECT productid, categoryid
FROM
  (SELECT_PHYSICAL productid, categoryid
   FROM "My_DB"."My_Schema"."products" a
   LEFT OUTER JOIN "My_DB"."My_Schema"."categories" b
   ON a.categoryid = b.categoryid);
```

```
SELECT y.cid, sum(x.qty)
FROM
  (SELECT productid pid, categoryid cid, qtysold qty
   FROM sales.product) x
RIGHT OUTER JOIN
  (SELECT_PHYSICAL CASE categoryid WHEN 1 THEN null ELSE categoryid END cid
```

```

FROM "My_DB"."My_Schema"."categories") y
ON x.cid = y.cid
GROUP BY y.cid;

```

- **Cross-Database Queries**

You can use `SELECT_PHYSICAL` to join tables in different databases. For example:

```

SELECT_PHYSICAL a.productid, b.categoryid
FROM "My_DB"."My_Schema"."products" a
FULL OUTER JOIN
"My_DB2"."My_Schema"."categories" b
ON a.categoryid = b.categoryid

```

Using the `NATURAL JOIN` Keyword

`SELECT_PHYSICAL` queries support the `NATURAL JOIN` syntax that enables using predefined join expressions.

For ADF data sources, the ViewLink in ADF becomes active. The `NATURAL JOIN` join type, however, is not exposed for use in Logical Table Sources, for example, `LEFT OUTER JOIN`.

You can only use the `NATURAL JOIN` keyword in `SELECT_PHYSICAL` queries. The `NATURAL JOIN` behavior in Oracle Business Intelligence is different from the ANSI `NATURAL JOIN`. The following examples illustrate how joins are executed with and without the `NATURAL JOIN` syntax:

```

SELECT PHYSICAL *
FROM A, B;

```

In this example, no join is executed between A and B, even if one is defined in the metadata.

```

SELECT_PHYSICAL *
FROM A NATURAL JOIN B;

```

In this example, the physical join between A and B is executed. For ADF data sources, the join expression defined by the underlying ViewLink is used.

```

SELECT_PHYSICAL *
FROM C, A NATURAL JOIN B;

```

In this example, even if C is joined to A in the metadata, only the A-B join is active. The C-A join is not used.

Special Usages of `SELECT_PHYSICAL`

You can use session variables and the `INDEXCOL` function in a `SELECT_PHYSICAL` command.

The following examples show the `INDEXCOL` function in a `SELECT_PHYSICAL` command:

```

SELECT_PHYSICAL VALUEOF(NQ_SESSION.REGION)
FROM "My_DB"."My_Schema"."products";

```

```
SELECT_PHYSICAL INDEXCOL(VALUEOF(NQ_SESSION.INDEXCOLINDEX), productid, categoryid)
FROM "My_DB"."My_Schema"."products";
```

Limiting and Offsetting Rows Returned

You can use the `FETCH` and `OFFSET` clauses to constrain the number of rows returned by the `SELECT` statement and to skip a specified number of rows from the beginning of the result set.

The `FETCH` and `OFFSET` clauses are optional. You can use the clauses together, or independently. The fetch and offset clauses are part of the `SELECT` statement and are placed at the end.

These clauses are useful for situations where you have a large result set such as with a large dimension, and you want to present, for example, the first 100 rows to the user. The Oracle BI Server stops processing when the limit is reached, improving overall performance and conserving resources. In addition, the limit is pushed to the back-end database in many cases so that the database can optimize the query.

You can use the `FETCH` and `OFFSET` clauses without an `ORDER BY` clause, but the results are non-deterministic. You should always use the `FETCH` and `OFFSET` clauses with an `ORDER BY` clause.

If `OFFSET` is not specified, the default value is 0, which means that results are returned starting from the first row. If `FETCH` is not specified, it means that there is no limitation on the number rows returned.

Both clauses are evaluated after the `WHERE` clause, aggregation, `HAVING` clause, window analytic function, and `ORDER BY` clause. You can use both clauses with `SELECT_PHYSICAL` in addition to `SELECT`.

Syntax for `OFFSET` Clause

```
OFFSET n ROW[S]
```

Where:

n is the number of rows you want to skip from the beginning of the result set. You must use a value in *n* that is greater than zero.

Syntax for `FETCH` Clause

```
FETCH FIRST | NEXT n ROW[S] ONLY
```

Where:

n is the number of rows you want to retrieve. You must use a value in *n* that is greater than zero.

Use the `FIRST` statement when the limit clause is used independently of the offset clause. Use `NEXT` when the limit clause is used in conjunction with the offset clause.

Example

```
SELECT employeeid, firstname, revenue
FROM sales.employee
ORDER BY revenue desc
OFFSET 2 ROWS
FETCH NEXT 4 ROWS ONLY
```

The following table lists the entire result set without the `OFFSET` and `FETCH` clauses. When the `OFFSET` and `FETCH` clauses are included, only the rows shown in bold are returned.

Employeeid	FirstName	Revenue
4	Margaret	250187.45
3	Janet	213051.30
1	Nancy	202143.71
2	Andrew	202143.71
7	Robert	177749.26
8	Laura	141295.99
9	Anne	133301.03
6	Michael	82964.00
5	Steven	78198.10

Limitations of the `FETCH` and `OFFSET` Clauses

Because `ORDER BY` clauses are ignored in `UNION ALL` set-operator blocks, using these clauses in such queries would be non-deterministic.

Do not use `FETCH` and `OFFSET` with these queries.

Rules for Queries with Aggregate Functions

The Oracle BI Server simplifies the SQL statements needed to craft aggregate queries.

This section outlines the rules that the Oracle BI Server follows for whether a query contains a `GROUP BY` clause and, if a `GROUP BY` clause is specified, what results you should expect from the query. The rules outlined in this section apply to all aggregates used in SQL statements (`SUM`, `AVG`, `MIN`, `MAX`, `COUNT (*)`, and `COUNT`).

Computing Aggregates of Baseline Columns

Baseline columns map to nonaggregated data at the level of granularity of the logical table to which they belong.

A baseline column does not have an aggregation rule defined in the Aggregation tab of the Logical Column dialog in the repository. If you perform aggregation (`SUM`, `AVG`, `MIN`, `MAX`, or `COUNT`) on a baseline column through a SQL request, the Oracle BI Server calculates the aggregation at the level based on the following rules:

- If there is no `GROUP BY` clause specified, the level of aggregation is grouped by all of the nonaggregate columns in the `SELECT` list.
- If there is a `GROUP BY` clause specified, the level of aggregation is based on the columns specified in the `GROUP BY` clause.

For example, consider the following query, where the column revenue is defined in the repository as a baseline column, no aggregation rules specified in the Logical Column > Aggregation tab:

```
SELECT year, product, SUM(revenue)
FROM time, products, facts
```

The results appear in the following list by year, products, and then sum of revenue.

YEAR	PRODUCT	SUM(REVENUE)
1998	Coke	500
1998	Pepsi	600
1999	Coke	600
1999	Pepsi	550
2000	Coke	800
2000	Pepsi	600

This query returns results grouped by year and product, or in other words, it returns one row for each product and year combination. The sum calculated for each row is the sum of all the sales for that product in that year. It is logically the same query as the following:

```
SELECT year, product, SUM(revenue)
FROM time, products, facts
GROUP BY year, product
```

If you change the GROUP BY clause to only group by year, then the sum calculated is the sum of all products for the year, as follows:

```
SELECT year, product, SUM(revenue)
FROM time, products, facts
GROUP BY year
```

The results appear in the following list by year, products, and then sum of revenue:

YEAR	PRODUCT	SUM(REVENUE)
1998	Coke	1100
1998	Pepsi	1100
1999	Coke	1150
1999	Pepsi	1150
2000	Coke	1400

YEAR	PRODUCT	SUM(REVENUE)
2000	Pepsi	1400

If you add a column to the query requesting the COUNT of revenue, the Oracle BI Server calculates the number of records used to calculate the results for each group. In this case, it is a year, as shown in the following example:

```
SELECT year, product, SUM(revenue), COUNT(revenue)
FROM time, products, facts
GROUP BY year
```

The results appear in the following list by year, products, sum of revenue, and then revenue count:

YEAR	PRODUCT	SUM(REVENUE)	COUNT(REVENUE)
1998	Coke	1100	6000
1998	Pepsi	1100	6000
1999	Coke	1150	6500
1999	Pepsi	1150	6500
2000	Coke	1400	8000
2000	Pepsi	1400	8000

Computing Aggregates of Measure Columns

Measure columns always calculate the aggregation with which they are defined.

A measure column has a default aggregation rule defined in the Aggregation tab of the Logical Column dialog in the repository. If you perform explicit aggregation (SUM, AVG, MIN, MAX, or COUNT) on a measure column through a SQL request, you are actually asking for an aggregate of an aggregate. For these nested aggregates, the Oracle BI Server calculates the aggregation based on the following rules:

- A request for a measure column without an aggregate function defined in a SQL statement is always grouped at the level of the nonaggregate columns in the SELECT list, regardless of whether the query specifies a GROUP BY clause.
- If there is no GROUP BY clause specified, the nested aggregate is a grand total of each group determined by all of the nonaggregate columns in the SELECT list.
- If there is a GROUP BY clause specified, the nested aggregation calculates the total for each group as specified in the GROUP BY clause.

For example, consider the following query, where the column SumOfRevenue is defined in the repository as a measure column with a default aggregation rule of SUM (SUM aggregation rule specified in the Aggregation tab of the Logical Column dialog):

```
SELECT year, product, SumOfRevenue, SUM(SumOfRevenue)
FROM time, products, facts
```

The following query results are grouped by year and product, or in other words, it returns one row for each product and year combination. The sum calculated for each row in the SumOfRevenue column is the sum of all the sales for that product in that year because the measure column is always at the level defined by the nonaggregation columns in the query.

YEAR	PRODUCT	SUMofREVENUE	SUM(SUMofREVENUE)
1998	Coke	500	3650
1998	Pepsi	600	3650
1999	Coke	600	3650
1999	Pepsi	550	3650
2000	Coke	800	3650
2000	Pepsi	600	3650

If you set the GROUP BY clause to only group by year, then the sum calculated in the SumOfRevenue column is the sum of each product for the year, and the sum calculated in the SUM(SumOfRevenue) column is total sales of all products for the given year. The following is the query:

```
SELECT year, product, SumOfRevenue, SUM(SumOfRevenue)
FROM time, products, facts
GROUP BY year
```

In the following result set, the sum calculated for each row in the SumOfRevenue column is the sum of all the sales for that product in that year because the measure column is always at the level defined by the nonaggregation columns in the query. The SUM(SumOfRevenue) is the same for each row corresponding to a given year, and that sum represents the total sales for that year. In this case, it is the sales of Coke plus the sales of Pepsi.

YEAR	PRODUCT	SUMofREVENUE	SUM(SUMofREVENUE)
1998	Coke	500	1100
1998	Pepsi	600	1100
1999	Coke	600	1150
1999	Pepsi	550	1150
2000	Coke	800	1400
2000	Pepsi	600	1400

Display Function Reset Behavior

A display function operates on the result set of a query.

The Oracle BI Server supports these display functions: RANK, TOPN, BOTTOMN, PERCENTILE, NTILE, MAVG, MEDIAN, and varieties of standard deviation when specified in the SQL query's SELECT list. Queries that use display functions conform to the following rules:

- If the GROUP BY clause is not specified, the display function operates across the entire result set, or in other words, the grouping level for the display function matches that of the query.
- If the GROUP BY clause is specified, the display function resets its values for each group as specified in the GROUP BY clause.

For example, in the following query, SumOfRevenue is defined as a measure column with the default aggregation rule of SUM:

```
SELECT year, product, SumOfRevenue, RANK(SumOfRevenue)
FROM time, products, facts
```

In the following query result set, the GROUP BY clause is not specified, so the rank is calculated across the entire result set:

YEAR	PRODUCT	SUMofREVENUE	RANK(SUMofREVENUE)
1998	Coke	500	6
1998	Pepsi	600	2
1999	Coke	600	2
1999	Pepsi	550	5
2000	Coke	800	1
2000	Pepsi	600	2

If you change the GROUP BY clause to group by year and product, then the rank is reset for each year, as follows:

```
SELECT year, product, SUM(revenue), RANK(sum(revenue) by year)
FROM time, products, facts
GROUP BY year, product
```

In the following result set, the rank is reset each time the year changes, and because there are two rows for each year, the value of the rank is always 1 or 2:

YEAR	PRODUCT	SUMofREVENUE	RANK(SUM(REVENUE) by year)
1998	Coke	500	2
1998	Pepsi	600	1

YEAR	PRODUCT	SUMofREVENUE	RANK(SUM(REVENUE) by year)
1999	Coke	600	1
1999	Pepsi	550	2
2000	Coke	800	1
2000	Pepsi	600	2

Alternative Syntax

You can calculate a specified level of aggregation using `BY` within the aggregate function.

When using an aggregate function, you can calculate a specified level of aggregation using `BY` within the aggregate function. If you use `BY`, you do not need a `GROUP BY` clause.

For example, the following query returns the column `year_revenue` that displays revenue aggregated by year:

```
SELECT year, product, revenue, SUM(revenue BY year) as year_revenue
FROM softdrinks
```

You can use the same syntax with display functions. The following query calculates overall rank of revenue for each product for each year (each row in the entire result set), and also the rank of each product's revenue within each year:

```
SELECT year, product, revenue, rank(revenue), RANK(revenue by year)
FROM softdrinks ORDER BY 1, 5
```

Using `FILTER` to Compute a Conditional Aggregate

`FILTER` is an operator that restricts the set of rows used to compute its aggregate argument to rows that satisfy the `USING` condition.

In SQL query language, traditional aggregates such as `SUM`, `COUNT`, `MIN`, and `MAX` are evaluated on a group of tuples, an ordered list of objects, each of a specified type, determined by the `GROUP BY` clause. All the aggregates specified in the `SELECT` clause of a query are evaluated over the same subset of tuples. Conditional aggregates extend SQL by restricting their input using a predicate.

The `FILTER` operator is a Logical SQL construct. You can use the `FILTER` operator in logical queries referring to the metadata, or in logical columns that use existing logical columns as the source.

Syntax

Conditional aggregates are only notational concepts and do not represent executable operators. Conditional aggregates are expressed in the form of a function as shown in the following statement:

```
FILTER(measure_expr USING boolean_expr)
```

Where:

measure_expr is an expression that contains at least one measure, for example:

- The expression `Sales + 1` is allowed if `Sales` is a measure.
- The expression `productid` is not allowed if `productid` is a scalar attribute.

boolean_expr is a boolean expression, evaluates to `TRUE` or `FALSE`, that does not contain any measures. This expression cannot contain any nested queries.

Example

The following is a simple example of the `FILTER` function:

```
SELECT year,
FILTER(sales USING product = 'coke'),
FILTER(sales USING product = 'pepsi')
FROM logBeverages
```

After navigation, this query is executed as follows:

```
SELECT year,
SUM(CASE WHEN product = 'coke' THEN sales),
SUM(CASE WHEN product = 'pepsi' THEN sales)
FROM physBeverages
WHERE product = 'coke' OR product = 'pepsi'
GROUP BY year
```

Error Handling

In the example `FILTER(x USING y)`, error messages are returned in the following situations:

- The *y* expression is not a boolean expression.
- The *y* expression contains measures.
- `FILTER` is used in outer query block.
- Explicit aggregates are used in the *x* (measure) expression. For example, `FILTER(COUNT(product), C)`.

Operators

SQL logical operators, and mathematical operators are two types of operators.

There are two types of operators: SQL logical operators, and mathematical operators.

SQL Logical Operators

The SQL logical operators are used to specify comparisons between expressions.

You can use the following SQL logical operators:

- `Between` is used to determine boundaries for a condition. Each boundary is an expression. The bounds do not include the boundary limits, as in less than and greater than, or as opposed to less than or equal to, and greater than or equal to. You can use `NOT` before `BETWEEN` to indicate that you are looking for results that are not between the two specified boundaries.
- `In`: Specifies a comparison of a column value with a set of values.
- `Is Null`: Specifies a comparison of a column value with the null value.

- **Like:** Specifies a comparison to a literal value. You can use `Like` with wildcard characters to indicate any character string match of zero or more characters (%) or a any single character match (_).

Mathematical Operators

Mathematical operators are used to combine expression elements to make certain types of comparisons in an expression.

The table lists available operators and describes their use in an expression.

Operator	Description
+	Plus sign for addition.
-	Minus sign for subtraction.
*	Multiply sign for multiplication.
/	Divide by sign for division.
	Character string concatenation.
(Open parenthesis.
)	Closed parenthesis.
>	Greater than sign, indicating values higher than the comparison.
<	Less than sign, indicating values lower than the comparison.
=	Equal sign, indicating the same value.
<=	Less than or equal to sign, indicating values the same or lower than the comparison.
>=	Greater than or equal to sign, indicating values the same or higher than the comparison.
<>	Not equal to, indicating values higher or lower, but different.
AND	AND connective, indicating intersection with one or more conditions to form a compound condition.
OR	OR connective, indicating the union with one or more conditions to form a compound condition.
NOT	NOT connective, indicating a condition is not met.
,	Comma, used to separate elements in a list.

Conditional Expressions

Expressions are building blocks for creating conditional expressions that convert a value from one form to another.

Expressions include:

- [CASE \(Switch\)](#)
- [CASE \(If\)](#)

CASE (Switch)

The CASE statement is a CASE(Lookup) form.

The value of *expr1* is examined, then the WHEN expressions. If *expr1* matches any WHEN expression, it assigns the value in the corresponding THEN expression.

If none of the WHEN expressions match, it assigns the default value specified in the ELSE expression. If no ELSE expression is specified, the system automatically adds an ELSE NULL.

If *expr1* matches an expression in multiple WHEN clauses, only the expression following the first match is assigned.

Note:

In a CASE statement, AND has precedence over OR.

Syntax

```
CASE expr1
  WHEN expr2 THEN expr3
  {WHEN expr... THEN expr...}
  ELSE expr
END
```

Where:

CASE starts the CASE statement. You must follow the CASE statement with an expression and one or more WHEN and THEN statements, an optional ELSE statement, and the END keyword.

WHEN specifies the condition to satisfy.

THEN specifies the value to assign if the corresponding WHEN expression is satisfied.

ELSE specifies the value to assign if none of the WHEN conditions are satisfied. If omitted, ELSE NULL is assumed.

END ends the CASE statement.

Example

```
CASE Score-par
  WHEN -5 THEN 'Birdie on Par 6'
  WHEN -4 THEN 'Must be Tiger'
  WHEN -3 THEN 'Three under par'
  WHEN -2 THEN 'Two under par'
  WHEN -1 THEN 'Birdie'
  WHEN 0 THEN 'Par'
  WHEN 1 THEN 'Bogey'
  WHEN 2 THEN 'Double Bogey'
```

```
    ELSE 'Triple Bogey or Worse'  
END
```

In this example, the WHEN statements must reflect a strict equality. For example, a WHEN condition of WHEN < 0 THEN 'Under Par' is illegal because comparison operators are not allowed.

CASE (If)

The CASE statement evaluates each WHEN condition and if satisfied, assigns the value in the corresponding THEN expression.

If none of the WHEN conditions are satisfied, it assigns the default value specified in the ELSE expression. If no ELSE expression is specified, the system automatically adds an ELSE NULL.

Note:

In a CASE statement, AND has precedence over OR.

Syntax

```
CASE  
    WHEN request_condition1 THEN expr1  
    {WHEN request_condition2 THEN expr2}  
    {WHEN request_condition... THEN expr...}  
    ELSE expr  
END
```

Where:

CASE starts the CASE statement. You must follow the CASE statement by one or more WHEN and THEN statements, an optional ELSE statement, and the END keyword.

WHEN specifies the condition to satisfy.

THEN specifies the value to assign if the corresponding WHEN expression is satisfied.

ELSE specifies the value to assign if none of the WHEN conditions are satisfied. If omitted, ELSE NULL is assumed.

END ends the CASE statement.

Example

```
CASE  
    WHEN score-par < 0 THEN 'Under Par'  
    WHEN score-par = 0 THEN 'Par'  
    WHEN score-par = 1 THEN 'Bogie'  
    WHEN score-par = 2 THEN 'Double Bogey'  
    ELSE 'Triple Bogey or Worse'  
END
```

Unlike the Switch form of the CASE statement, the WHEN statements in the If form allow comparison operators. For example, a WHEN condition of WHEN < 0 THEN 'Under Par' is legal.

Expressing Literals

Learn how to express each type of literal in SQL.

A literal is a nonnull value corresponding to a given data type. Literals are typically constant values, or in other words, they are values that are taken as they are. A literal value must comply with the data type that it represents.

SQL provides mechanisms for expressing literals in SQL statements. The following topics describe how to express each type of literal in SQL:

- [Character Literals](#)
- [Datetime Literals](#)
- [Numeric Literals](#)

Character Literals

A character literal represents a value of CHARACTER or VARCHAR data type.

To express a character literal, enclose the character string in single quotes ('). The number of characters enclosed between the single quotes implies the length of the literal.

Example

```
'Oracle BI Server'  
  
'abc123'
```

Datetime Literals

Learn about three *typed* datetime literals.

The SQL 92 standard defines three kinds of *typed* datetime literals, in the following formats:

```
DATE 'yyyy-mm-dd'  
TIME 'hh:mm:ss'  
TIMESTAMP 'yyyy-mm-dd hh:mm:ss'
```

To express a typed datetime literal, use the keywords DATE, TIME, or TIMESTAMP followed by a datetime string enclosed in single quotation marks, as in the preceding example. Two digits are required for all nonyear components even if the value is a single digit.

These formats are fixed and are not affected by the format specified in the NQSConfig.INI file for the parameters DATE_DISPLAY_FORMAT, TIME_DISPLAY_FORMAT, or DATE_TIME_DISPLAY_FORMAT.

Examples

```
DATE '2000-08-15'  
TIME '11:55:25'  
TIMESTAMP '1999-03-15 11:55:25'
```

Numeric Literals

A numeric literal represents a value of a numeric data type such as `INTEGER`, `DECIMAL`, or `FLOAT`.

To express a numeric literal, type the number as part of a SQL statement.

Do not surround numeric literals with single quotes. Using single quotes expresses the literal as a character literal.

Note:

When `ENABLE_NUMERIC_DATA_TYPE` is set to `YES` in `NQSConfig.INI`, all decimal literals or integer literals that are too large to fit in the `INT` data type are parsed internally within the Oracle BI Server as `NUMERIC`.

When treating literals as `NUMERIC`, consider the Oracle standard double promotion rules that include the following:

`DOUBLE/NUMBER = DOUBLE` , `DOUBLE * NUMBER = DOUBLE`

Because the parsing of numeric literals happens very early in the query processing before the actual data source is known, internally, the Oracle BI Server treats decimal numbers as `NUMERIC` if `ENABLE_NUMERIC_DATA_TYPE` is set to `YES`, regardless of data source type.

When `NUMERIC` is enabled and the Oracle BI Server executes an expression internally involving decimal literals, the server treats the literals as `NUMERIC` even if the back-end data source does not support the `NUMERIC` data type. However, the type promotion rules still apply. For example, if the Oracle BI Server retrieves the data from a data source as `DOUBLE` and combines that with a `NUMERIC` literal during internal execution, the final result converted to `DOUBLE`.

Numeric literals include:

- [Integer Literals](#)
- [Decimal Literals](#)
- [Floating Point Literals](#)

Integer Literals

To express an integer constant as a literal, specify the integer as part of a SQL statement, for example, in the `SELECT` list.

Precede the integer with a plus sign (+) to indicate the integer is positive, or a minus sign (-) to indicate the integer is negative. Unsigned integers are treated as positive.

Examples

```
234
+2
567934
```

Decimal Literals

Specify a decimal number to express a decimal literal.

Precede the number with a plus sign (+) to indicate the number is positive, or a minus sign (-) to indicate the number is negative. Unsigned numbers are treated as positive numbers.

Examples

```
1.223
-22.456
+33.456789
```

Floating Point Literals

To express floating point numbers as literal constants, enter a decimal literal followed by the letter E using uppercase or lowercase, followed by the plus sign (+) to indicate a positive exponent, or the minus sign (-) to indicate a negative exponent.

No spaces are allowed between the integer, the letter E, and the sign of the exponent.

Examples

```
333.456E-
1.23e+
```

Calculated Members

A calculated member is a user-defined dimension member whose measure values are calculated at run time.

You define a calculated member within a dimension through a formula that references other members of the same dimension. If a dimension has multiple hierarchies, all members referenced in the formula must belong to one hierarchy.

Within a calculated member, the members can exist at different levels in the hierarchy. For example, in a Geography hierarchy, you can create a calculated member to enable you to add together measure values for the Country member France and the City member Budapest.

The three standard components of a calculated member are:

- The presentation hierarchy on which the calculated member is based, for example, *Geography*
- The name to identify the calculated member, and to distinguish it from other members in the dimension, for example, *My Locations*
- The formula used to calculate the calculated member, containing one or more Member clauses, for example, Member ("Geography"."Country".'France') + Member ("Geography"."City".'Budapest')

This section contains the following topics:

- [CALCULATEDMEMBER Syntax](#)
- [Rules for the CALCULATEDMEMBER Expression](#)
- [Using Solve Order to Control Formula Evaluation Sequence](#)
- [Examples of Calculated Members in Queries](#)

CALCULATEDMEMBER Syntax

The CALCULATEDMEMBER function syntax includes rules for use with presentation hierarchies and presentation hierarchy levels.

```
CALCULATEDMEMBER(presentation_hierarchy, member_identifier,  
calculated_member_formula [, solve_order])
```

Where:

presentation_hierarchy identifies the fully qualified presentation hierarchy in the presentation layer on which the calculated member is based, as follows:

```
"subject_area"."presentation_table"."presentation_hierarchy"
```

Note:

When qualifying presentation hierarchies and presentation hierarchy levels in both the CALCULATEDMEMBER expression and the Member clause within the *calculated_member_formula* parameter, the following rule applies:

- You must specify the qualification term ("subject_area" .) if there are multiple presentation tables or presentation hierarchies with the same name in different subject areas, otherwise you can omit the term.

member_identifier is the string or numeric literal that identifies the calculated member. The type of literal depends on the data type of the dimension level-keys.

calculated_member_formula consists of one or more examples of a "member clause" connected by the standard arithmetic operators + - * / (). The syntax of the member clause depends on whether the presentation hierarchy is level-based or parent-child. See [Syntax for the Member Clause in Level-Based Hierarchies](#) and [Syntax for the Member Clause in Parent-Child Hierarchies](#) for details.

solve_order (*optional*) is a positive integer, used to determine the order of evaluation when there are calculated members from different dimensions in the same query. See [Using Solve Order to Control Formula Evaluation Sequence](#) for details.

Syntax for the Member Clause in Level-Based Hierarchies

```
MEMBER(presentation_hierarchy_level, member_value)
```

Where:

presentation_hierarchy_level identifies the fully qualified hierarchy level in the *presentation_hierarchy*, as follows:

```
"subject_area"."presentation_table"."presentation_hierarchy"."presentation_level"
```

member_value is the string or numeric literal that identifies the member in the *presentation_hierarchy_level*.

Syntax for the Member Clause in Parent-Child Hierarchies

```
MEMBER(presentation_hierarchy, member_value)
```

Where:

presentation_hierarchy identifies the fully qualified presentation hierarchy in the presentation layer on which the calculated member is based, as follows:

```
"subject_area"."presentation_table"."presentation_hierarchy"
```

member_value is the string or numeric literal that identifies the member in the *presentation_hierarchy*.

Rules for the CALCULATEDMEMBER Expression

The rules for calculated members relate to the CALCULATEDMEMBER expression, and the CALCULATEDMEMBER expression when used in queries.

- All level references in a given CALCULATEDMEMBER expression must belong to the same dimension hierarchy.
- CALCULATEDMEMBER expressions may only appear in the SELECT list of a query.
- Queries containing a CALCULATEDMEMBER expression must include at least one measure column in the SELECT list.
- Only one CALCULATEDMEMBER expression is allowed for each dimension for each SELECT list of a query block. However, CALCULATEDMEMBER expressions based on other dimensions may exist in the same query.
- You cannot include any other column from a dimension on which a calculated member is based in the following components of a query block:
 - SELECT list
 - WHERE clause
 - HAVING clause

However, you may reference columns from the calculated member dimension in subqueries.

- Columns from other dimensions may be referenced in the same query block, as long as there are no CALCULATEDMEMBER expressions on those dimensions.

Using Solve Order to Control Formula Evaluation Sequence

By default, when the CALCULATEDMEMBER expression does not contain a solve order, the calculated members are evaluated in the order in which they appear in the SELECT list.

When there are calculated members from different dimensions in the same query block, the order in which the Oracle BI Server evaluates the calculated members may be significant.

The following illustrates how the wrong solve order can lead to incorrect results.

Using Solve Order

Assume you have the following account and time data:

Kwik Grains	Profit	Sales
2007 Q3	300	1000
2007 Q4	600	1500

You want to calculate the percentage profit (Profit / Sales * 100) for each time period and the totals for the two quarters.

If the solve order for your calculations is the following:

1. 'Profit%' = 'Profit'/'Sales' * 100
2. '2007 Second Half' = '2007 Q3' + '2007 Q4'

then the percentage profit for '2007 Second Half' is calculated *incorrectly* by adding the 2007 Q3 profit to the 2007 Q4 profit:

- $(300/1000) + (600/1500) = 30\% + 40\% = 70\%$

with the following results:

Kwik Grains	Profit	Sales	Profit%
2007 Q3	300	1000	30
2007 Q4	600	1500	40
2007 Second Half	900	2500	70 (<i>incorrect result</i>)

If the solve order for your calculations is the following:

1. '2007 Second Half' = '2007 Q3' + '2007 Q4'
2. 'Profit%' = 'Profit'/'Sales' * 100

then the percentage profit for '2007 Second Half' is calculated correctly by adding the 2007 Q3 and 2007 Q4 profits and sales first, then dividing the total Profit by the total Sales:

- $(300+600) / (1000+1500) = 900/2500 = 36\%$

with the following results:

Kwik Grains	Profit	Sales	Profit%
2007 Q3	300	1000	30
2007 Q4	600	1500	40
2007 Second Half	900	2500	36 (<i>correct result</i>)

See [Example 1-2](#) for an example of a query that explicitly specifies the solve order.

Examples of Calculated Members in Queries

These examples show the use of calculated members in queries, and the base data on which the calculations are performed.

Example 1-1 Single Calculated Member Query

Single Calculated Member Query

This example shows two queries, each with corresponding results.

The first query contains a calculated member.

```
SELECT CALCULATEDMEMBER(product."Product - Region", 'USA - LA - Tokyo',
    MEMBER(product."Product - Region"."Country", 'USA')
    - MEMBER(product."Product - Region"."Region", 'LA')
    - MEMBER(product."Product - Region"."City", 'Tokyo')
    ) MyRegion,
    sales.Revenue Revenue, sales.QtySold QtySold
FROM product, sales;
```

Result:

MYREGION	REVENUE	QTYSOLD
USA - LA - Tokyo	61959.00	3959

The second query verifies the results of the first query. It shows the base data on which the calculation in the first query is performed.

```
SELECT * from SupplierCity where Country in ('USA', 'Japan');
```

Result:

CITY	REGION	COUNTRY	REVENUE	QTYSOLD
Boston	MA	USA	28146.40	2084
Osaka		Japan	15678.30	1417
New Orleans	LA	USA	33351.95	1735
Ann Arbor	MI	USA	43569.00	1436
Tokyo		Japan	33533.20	1134
Bend	OR	USA	23776.80	1573

Example 1-2 Using Calculated Members from Different Dimensions

Using Calculated Members from Different Dimensions

The requirement in this example is to determine the percentage increase over time in Revenue and Quantity Sold for US and Canada combined.

To achieve the correct results, the solve order is significant. You must first add Revenue and Quantity Sold for the two countries across the time periods, then perform the percentage calculation. See [Using Solve Order to Control Formula Evaluation Sequence](#) for more information about solve order significance.

This example shows two queries, each with corresponding results.

The first query contains the calculated members from the two dimensions Product - Region and Time, with the "addition" formula calculated first, then the "percentage" formula.

```
SELECT CALCULATEDMEMBER(product."Product - Region", 'North America',
    MEMBER(product."Product - Region"."Country", 'USA')
    + MEMBER(product."Product - Region"."Country", 'Canada'), 1
    ) MyRegion,
    CALCULATEDMEMBER(day."Time", 'Percentage Increase',
```

```

        ( MEMBER(day."Time"."Year", 1996)
        - MEMBER(day."Time"."Year", 1995) ) * 100
        / MEMBER(day."Time"."Year", 1995), 2
    ) MyTime,
    sales.Revenue RevenuePC,
    sales.QtySold QtySoldPC
FROM product, sales, day;

```

Result:

MYREGION	MYTIME	REVENUEPC	QTYSOLDPC
North America	Percentage Increase	16	35

Note that in the preceding query, the sequence of the calculated members in the SELECT list is sufficient for correct results, even without the explicit solve orders. The solve orders are included for completeness.

The second query verifies the results of the first query. It shows the base data on which the calculations in the first query are performed.

```

SELECT CALCULATEDMEMBER(product."Product - Region", 'North America',
    MEMBER(product."Product - Region"."Country", 'USA')
    + MEMBER(product."Product - Region"."Country", 'Canada'))
    MyRegion,
    year as Year, sales.Revenue Revenue, sales.QtySold QtySold
FROM product, sales, day;

```

Result:

MYREGION	YEAR	REVENUE	QTYSOLD
North America	1996	101702.75	4918
North America	1995	87265.10	3638
North America	1994	30776.00	1616

Variables

You can include and set variables in SQL statements.

To do this, include the variable at the beginning of the SQL statement.

Syntax

```
SET VARIABLE variable_name = variable_value; SELECT_statement
```

If you are executing a query from the nqcmd utility, use a colon as a delimiter. Otherwise, you can use either a semicolon or a colon.

Examples

```
SET VARIABLE LOGLEVEL = 3; SELECT Products.Brand, Measures.Dollars FROM "Products"
```

```
SET VARIABLE DISABLE_CACHE_HIT=1, LOGLEVEL = 3, WEBLANGUAGE='en': SELECT
Products.Brand, Measures.Dollars FROM "Products"
```

Aggregate, Running Aggregate, Time Series, and Reporting Functions

These sections contain information about aggregate functions, running aggregate functions, and time series functions.

- [Aggregate Functions](#)
- [Running Aggregate Functions](#)
- [Time Series Functions](#)
- [Reporting Functions](#)

Aggregate Functions

Aggregate functions perform operations on multiple values to create summary results.

The aggregate functions cannot be used to form nested aggregation in expressions on logical columns that have a default aggregation rule defined in the Aggregation tab of the Logical Column dialog. To specify nested aggregation, you must define a column with a default aggregation rule and then request the aggregation of the column in a SQL statement.

Aggregate functions include:

- [AGGREGATE AT](#)
- [AVG](#)
- [AVGDISTINCT](#)
- [BOTTOMN](#)
- [COUNT](#)
- [COUNTDISTINCT](#)
- [COUNT\(*\)](#)
- [FIRST](#)
- [FIRST_PERIOD](#)
- [GROUPBYCOLUMN](#)
- [GROUPBYLEVEL](#)
- [LAST](#)
- [LAST_PERIOD](#)
- [MAX](#)
- [MEDIAN](#)
- [MIN](#)
- [NTILE](#)
- [PERCENTILE](#)

- RANK
- STDDEV
- STDDEV_POP
- SUM
- SUMDISTINCT
- TOPN

AGGREGATE AT

The Aggregate At function summarizes the level specified with the AT keyword.

Syntax

This function aggregates columns based on the level or levels you specify. Using `AGGREGATE AT` guarantees that the aggregate for the measure always occurs at the levels specified after the keyword `AT`, regardless of the `WHERE` clause.

```
AGGREGATE(expr AT level [, level1, levelN])
```

Where:

expr is any expression that references at least one measure column

level is the level at which you want to aggregate. You can optionally specify multiple levels.

You cannot specify a level from a dimension that contains levels that are being used as the measure level for the measure you specified in the first argument. For example, you cannot write the function as `AGGREGATE(yearly_sales AT month)` because "month" is from the same time dimension that is being used as the measure level for "yearly_sales."

Example

The following example shows the `AGGREGATE AT` function and example results:

```
SELECT month, year, AGGREGATE(sales AT Year)
FROM timeseriestesting
WHERE year = 1994 AND month = 12
```

Result:

Month	Year	AGGREGATE AT year
12	1994	7396

Row count: 1

Because the `AGGREGATE AT` operator is always executed before the predicates, it always returns the correct total for the time level specified after the keyword `AT`.

AVG

The `AVG` function calculates the average (mean) value of an expression in a result set.

The `AVG` function must take a numeric expression as its argument.

Note that the denominator of AVG is the number of rows aggregated. For this reason, it is usually a mistake to use `AVG(x)` in a calculation in Oracle Business Intelligence. Instead, write the expression manually so that you can control both the numerator and denominator (x/y).

Syntax

`AVG(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

AVGDISTINCT

The AVGDISTINCT function calculates the average (mean) of all distinct values of an expression.

It must take a numeric expression as its argument.

Syntax

`AVG(DISTINCT numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

BOTTOMN

The BOTTOMN function ranks the lowest *n* values of the expression argument from 1 to *n*, 1 corresponding to the lowest numeric value.

The BOTTOMN function operates on the values returned in the result set. A request can contain only one BOTTOMN expression.

Syntax

`BOTTOMN(numExpr, integer)`

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer. Represents the bottom number of rankings displayed in the result set, 1 being the lowest rank.

COUNT

The COUNT function calculates the number of rows having a nonnull value for the expression.

The expression is typically a column name, in which case the number of rows with nonnull values for that column is returned.

Syntax:

`COUNT(expr)`

Where:

expr is any expression.

COUNTDISTINCT

The COUNTDISTINCT function computes the results using distinct processing in the COUNT function.

Syntax

```
COUNT(DISTINCT expr)
```

Where:

expr is any expression.

COUNT(*)

The COUNT(*) function computes the number of rows.

Syntax

```
COUNT(*)
```

Example

For example, if a table named Facts contained 200,000,000 rows, the sample request would return the results shown:

```
SELECT COUNT(*) FROM Facts
```

Result:

```
200000000
```

FIRST

The FIRST function selects the first non-null value of the expression argument.

Do not use the FIRST function when you need to compute the first value based on the chronological key rather than the primary level key, or if you need to return the first value regardless of whether it is null. Instead, use the FIRST_PERIOD function. See [FIRST_PERIOD](#) for more information.

The FIRST function is limited to defining dimension-specific aggregation rules in a repository. You cannot use it in SQL statements.

The FIRST function operates at the most detailed level specified in your explicitly defined dimension. For example, if you have a time dimension defined with hierarchy levels day, month, and year, the FIRST function returns the first day in each level.

You should not use the FIRST function as the first dimension-specific aggregate rule. It might cause queries to return large numbers of rows for processing in the Oracle BI Server, causing poor performance.

When a measure is based on dimensions, and data is dense, the Oracle BI Server optimizes the SQL statements sent to the database to improve performance. See *Setting Default Levels of Aggregation for Measure Columns for Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* more information about dense data.

Note:

You cannot nest `PERIODROLLING`, `FIRST`, `FIRST_PERIOD`, `LAST`, and `LAST_PERIOD` functions.

Syntax

`FIRST(expr)`

Where:

`expr` is any expression that references at least one measure column.

Example

`FIRST(sales)`

FIRST_PERIOD

The `FIRST_Period` function selects the first returned value of the expression argument.

The `FIRST_PERIOD` function can calculate the value of the first day of the year.

The `FIRST` and `FIRST_PERIOD` functions have different semantics. `FIRST` returns the first non-missing, that is recorded value of its argument in the given period. The `FIRST_PERIOD` returns the value that corresponds to the first chronological key in the given period, or nothing if that value is missing.

For example, consider the following date and sales table:

Date	Inventory
Jan 5	10
Jan 10	20

If `Inventory` is a measure with the aggregation rule `FIRST`, the value in query `SELECT Month, Inventory` equals 10, that is, the first recorded value.

If `Inventory` is a measure with aggregation rule `FIRST_PERIOD`, the value for the same query is an empty set because there is no recorded `Inventory` on January 1, assuming you are using a standard calendar table.

Note:

The following `FIRST_PERIOD` function restrictions:

- The `FIRST_PERIOD` function is limited to defining dimension-specific aggregation rules in a repository.
- When using the `FIRST_PERIOD` function, you must select the **Data is dense** field on the Logical Column Aggregation tab in the Administration Tool. If you do not select this field, then the aggregation rule reverts to the `FIRST` function.
- You cannot use the `FIRST_PERIOD` function in SQL statements.

- The `FIRST_PERIOD` function is only applicable to dimensions marked as time dimensions. Use the `FIRST` function for other dimensions.
 - Do not use the `FIRST_PERIOD` function when defining a dimension dependent measure on more than one time dimension. Instead, use the `FIRST` function when defining dimension-specific aggregation rules.
 - Do not use the `FIRST_PERIOD` function as the first dimension-specific aggregate rule. It might cause queries to bring back large numbers of rows for processing in the Oracle BI Server, causing poor performance.
 - You cannot nest `PERIODROLLING`, `FIRST`, `FIRST_PERIOD`, `LAST`, and `LAST_PERIOD` functions.
-

The `FIRST_PERIOD` function operates at the most detailed level specified in your explicitly defined dimension. For example, if you have a time dimension defined with hierarchy levels day, month, and year, the `FIRST_PERIOD` function returns the first day in each level.

When a measure is based on dimensions, and data is dense, the Oracle BI Server optimizes the SQL statements sent to the database to improve performance. See *Setting Default Levels of Aggregation for Measure Columns in Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* for more information about dense data.

Syntax

```
FIRST_PERIOD(expr)
```

Where:

expr is any expression that references at least one measure column.

```
FIRST_PERIOD(sales)
```

GROUPBYCOLUMN

The `GROUPBYCOLUMN` is for use in setting up aggregate navigation.

It specifies the logical columns that define the level of the aggregate data existing in a physical aggregate table.

For example, if an aggregate table contains data grouped by store and by month, specify the following syntax in the content filter in the General tab of Logical Source dialog.

```
GROUPBYCOLUMN(STORE, MONTH)
```

The `GROUPBYCOLUMN` function is only for use in configuring a repository. You cannot use it to form SQL statements.

GROUPBYLEVEL

`GROUPBYLEVEL` is for use in setting up aggregate navigation.

It specifies the dimension levels that define the level of the aggregate data existing in a physical aggregate table.

For example, if an aggregate table contains data at the store and month levels, and if you have defined dimensions (Geography and Customers) that contains these levels, specify the following syntax in the content filter in the General tab of Logical Source dialog.

```
GROUPBYLEVEL(GEOGRAPHY.STORE, CUSTOMERS.MONTH)
```

The `GROUPBYLEVEL` function is only for use in configuring a repository. You cannot use it to form SQL statements.

LAST

The `LAST` function selects the last non-null value of the expression.

Do not use the `LAST` function when you need to compute the last value based on the chronological key rather than the primary level key, or if you need to return the last value regardless of whether it is null. Instead, use the `LAST_PERIOD` function. See [LAST_PERIOD](#) for more information.

The `LAST` function is limited to defining dimension-specific aggregation rules in a repository. You cannot use it in SQL statements.

The `LAST` function operates at the most detailed level specified in your explicitly defined dimension. For example, if you have a time dimension defined with hierarchy levels day, month, and year, the `LAST` function returns the last day in each level.

You should not use the `LAST` function as the first dimension-specific aggregate rule. It might cause queries to bring back large numbers of rows for processing in the Oracle BI Server, causing poor performance.

When a measure is based on dimensions, and data is dense, the Oracle BI Server optimizes the SQL statements sent to the database to improve performance. See *Setting Default Levels of Aggregation for Measure Columns in Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* for more information about dense data.

Note that you cannot nest `PERIODROLLING`, `FIRST`, `FIRST_PERIOD`, `LAST`, and `LAST_PERIOD` functions.

Syntax

```
LAST(expr)
```

Where:

expr is any expression that references at least one measure column.

Example

```
LAST(sales)
```

LAST_PERIOD

The `LAST_PERIOD` function selects the last returned value of the expression.

For example, the `LAST_PERIOD` function can calculate the value of the last day of the year.

The `LAST_PERIOD` function operates at the most detailed level specified in your explicitly defined dimension. For example, if you have a time dimension defined with

hierarchy levels day, month, and year, the `LAST_PERIOD` function returns the last day in each level.

The `LAST` and `LAST_PERIOD` functions have different semantics. `LAST` returns the last non-missing, that is, the recorded value of its argument in the given period. Whereas `LAST_PERIOD` returns either the value that corresponds to the last chronological key in the given period or nothing if that value is missing.

For example, consider the following date and sales table:

Date	Inventory
Jan 1	10
Jan 5	20

If `Inventory` is a measure with the aggregation rule `LAST`, the value in query `SELECT Month, Inventory` is 20, that is, the last recorded value.

If `Inventory` is a measure with aggregation rule `LAST_PERIOD`, the value for the same query is an empty set because there was no recorded `Inventory` on January 31, assuming you are using a standard calendar table.

Note the following `LAST_PERIOD` function restrictions:

- The `LAST_PERIOD` function is limited to defining dimension-specific aggregation rules in a repository.
- When using the `LAST_PERIOD` function, you must select the **Data is dense** field on the Logical Column Aggregation tab in the Oracle BI Administration Tool. If you do not select this field, then the aggregation rule reverts to the `LAST` function.
- You cannot use the `LAST_PERIOD` function in SQL statements.
- The `LAST_PERIOD` function is only applicable to time dimensions. Use the `LAST` function for other dimensions.
- Do not use the `LAST_PERIOD` function when defining a dimension dependent measure on more than one time dimension. Instead, use the `LAST` function when defining dimension-specific aggregation rules.
- Do not use the `LAST_PERIOD` function as the first dimension-specific aggregate rule. It might cause queries to bring back large numbers of rows for processing in the Oracle BI Server, causing poor performance.
- You cannot nest `PERIODROLLING`, `FIRST`, `FIRST_PERIOD`, `LAST`, and `LAST_PERIOD` functions.

When a measure is based on dimensions, and data is dense, the Oracle BI Server optimizes the SQL statements sent to the database to improve performance. See *Setting Default Levels of Aggregation for Measure Columns in Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* for more information about dense data.

Syntax

`LAST_PERIOD(expr)`

Where:

expr is any expression that references at least one measure column.

Example

```
LAST_PERIOD(sales)
```

MAX

The MAX function calculates the maximum value (highest numeric value) of the rows satisfying the numeric expression argument.

Syntax

```
MAX(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

The MAX function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

MEDIAN

The MEDIAN function calculates the median (middle) value of the rows satisfying the numeric expression argument.

When there are an even number of rows, the median is the mean of the two middle rows. This function always returns a double.

Syntax

```
MEDIAN(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

The MEDIAN function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

MIN

The MIN function calculates the minimum value (lowest numeric value) of the rows satisfying the numeric expression argument.

Syntax

```
MIN(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

The MIN function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

NTILE

The NTILE function determines the rank of a value in terms of a user-specified range.

It returns integers to represent any range of ranks. In other words, the resulting sorted data set is broken into several tiles where there are roughly an equal number of values in each tile.

`NTile` with `numTiles = 100` returns what is commonly called the *percentile* with numbers ranging from 1 to 100, with 100 representing the high end of the sort. This value is different from the results of the Oracle BI EE `PERCENTILE` function, conforms to the percent rank in SQL 92, and returns values from 0 to 1.

Syntax

```
NTILE(numExpr, numTiles)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

`numTiles` is a positive, nonnull integer that represents the number of tiles.

If the `numExpr` argument is not null, the function returns an integer that represents a rank within the requested range.

PERCENTILE

The `PERCENTILE` function calculates a percent rank for each value satisfying the numeric expression argument.

The percentile rank ranges are from 0 (1st percentile) to 1 (100th percentile), inclusive.

The percentile is calculated based on the values in the result set.

Syntax

```
PERCENTILE(numExpr)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

The `PERCENTILE` function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

RANK

The `RANK` function calculates the level for each value satisfying the numeric expression argument.

The `RANK` function calculates the rank for each value satisfying the numeric expression argument. The highest number is assigned a rank of 1, and each successive rank is assigned the next consecutive integer (2, 3, 4,...). If certain values are equal, they are assigned the same rank (for example, 1, 1, 1, 4, 5, 5, 7...).

The rank is calculated based on the values in the result set.

Syntax

```
RANK(numExpr)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

The RANK function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

STDDEV

The STDDEV function returns the standard deviation for a set of values.

The return type is always a double. `STDDEV_SAMP` is a synonym for `STDDEV`.

Syntax

```
STDDEV([ALL | DISTINCT] numExpr)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

If `ALL` is specified, the standard deviation is calculated for all data in the set.

If `DISTINCT` is specified, all duplicates are ignored in the calculation.

If nothing is specified (the default), all data is considered.

The STDDEV function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

STDDEV_POP

The STDDEV_POP function returns the standard deviation for a set of values using the computational formula for population variance and standard deviation.

Syntax

```
STDDEV_POP([ALL | DISTINCT] numExpr)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

If `ALL` is specified, the standard deviation is calculated for all data in the set.

If `DISTINCT` is specified, all duplicates are ignored in the calculation.

If nothing is specified (the default), all data is considered.

SUM

The SUM function calculates the amount obtained by adding up all values satisfying the numeric expression argument.

Syntax

```
SUM(numExpr)
```

Where:

`numExpr` is any expression that evaluates to a numeric value.

The SUM function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

SUMDISTINCT

The SUMDISTINCT function calculates the amount obtained by adding all of the distinct values satisfying the numeric expression argument.

Syntax

```
SUM(DISTINCT numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

TOPN

The TOPN function ranks the highest *n* values of the expression argument from 1 to *n*, 1 corresponding to the highest numeric value.

The TOPN function operates on the values returned in the result set. A request can contain only one TOPN expression.

Syntax

```
TOPN(numExpr, integer)
```

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer. Represents the top number of rankings displayed in the result set, 1 being the highest rank.

The TOPN function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

Running Aggregate Functions

Running aggregate functions are similar to functional aggregates in that they take a set of records as input, but instead of outputting the single aggregate for the entire set of records, they output the aggregate based on records encountered so far.

This section describes the running aggregate functions supported by the Oracle BI Server. Functions include:

- [MAVG](#)
- [MSUM](#)
- [RSUM](#)
- [RCOUNT](#)
- [RMAX](#)
- [RMIN](#)

MAVG

The MAVG function calculates a moving average (mean) for the last *n* rows of data in the result set, inclusive of the current row.

The average for the first row is equal to the numeric expression for the first row. The average for the second row is calculated by taking the average of the first two rows of data. The average for the third row is calculated by taking the average of the first three rows of data, and so on until you reach the n th row, where the average is calculated based on the last n rows of data.

Syntax

```
MAVG(numExpr, integer)
```

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer. Represents the average of the last n rows of data.

The MAVG function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

MSUM

The MSUM function calculates a moving sum for the last n rows of data, inclusive of the current row.

The sum for the first row is equal to the numeric expression for the first row. The sum for the second row is calculated by taking the sum of the first two rows of data. The sum for the third row is calculated by taking the sum of the first three rows of data, and so on. When the n th row is reached, the sum is calculated based on the last n rows of data.

Syntax

```
MSUM(numExpr, integer)
```

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer. Represents the average of the last n rows of data.

The MSUM function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

Example

This example shows a query that uses the MSUM function, along with example query results.

```
select month, revenue, MSUM(revenue, 3) as 3_MO_SUM from sales_subject_area
```

Result:

MONTH	REVENUE	3_MO_SUM
JAN	100.00	100.00
FEB	200.00	300.00
MAR	100.00	400.00
APRIL	100.00	400.00
MAY	300.00	500.00
JUNE	400.00	800.00

JULY	500.00	1200.00
AUG	500.00	1400.00
SEPT	500.00	1500.00
OCT	300.00	1300.00
NOV	200.00	1000.00
DEC	100.00	600.00

RSUM

The RSUM function calculates a running sum based on records encountered so far.

The sum for the first row is equal to the numeric expression for the first row. The sum for the second row is calculated by taking the sum of the first two rows of data. The sum for the third row is calculated by taking the sum of the first three rows of data, and so on.

Syntax

```
RSUM(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

The RSUM function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

In Answers, you can also use the following alternate syntax:

```
RSUM(expression1 [BY expression2[, expression3[, ...]]])
```

Where:

expression1, *expression2*, *expression3* ... can represent any column reference, or an arithmetic expression on column references.

The BY clause causes the RSUM computation to restart at the row where the value of any of the BY columns differs from the previous row.

Example

This example shows a query that uses the RSUM function, along with example query results.

```
SELECT month, revenue, RSUM(revenue) as RUNNING_SUM from sales_subject_area
```

Result:

MONTH	REVENUE	RUNNING_SUM
JAN	100.00	100.00
FEB	200.00	300.00
MAR	100.00	400.00
APRIL	100.00	500.00
MAY	300.00	800.00
JUNE	400.00	1200.00
JULY	500.00	1700.00
AUG	500.00	2200.00
SEPT	500.00	2700.00

OCT	300.00	3000.00
NOV	200.00	3200.00
DEC	100.00	3300.00

RCOUNT

The RCOUNT function takes a set of records as input and counts the number of records encountered so far.

Syntax

RCOUNT(*expr*)

Where:

expr is an expression of any data type.

The RCOUNT function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

In Answers, you can also use the following alternate syntax:

RCOUNT(*expression1* [BY *expression2*[, *expression3*[, ...]]])

Where:

expression1, *expression2*, *expression3* ... can be any column reference, or an arithmetic expression on column references.

The BY clause causes the RCOUNT computation to restart at the row where the value of any of the BY columns differs from the previous row.

Example

This example shows a query that uses the RCOUNT function, along with example query results.

```
select month, profit, RCOUNT(profit) from sales_subject_area where profit > 200
```

Result:

MONTH	PROFIT	RCOUNT(profit)
MAY	300.00	2
JUNE	400.00	3
JULY	500.00	4
AUG	500.00	5
SEPT	500.00	6
OCT	300.00	7

RMAX

The RMAX function takes a set of records as input and shows the maximum value based on records encountered so far.

The specified data type must be one that can be ordered.

Syntax

RMAX(*expr*)

Where:

expr is of any data type. You must use a data type one that has an associated sort order.

The RMAX function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

In Answers, you can also use the following alternate syntax in Oracle BI Answers:

```
RMAX(expression1 [BY expression2[, expression3[, ...]]])
```

Where:

expression1, *expression2*, *expression3* ... can be any column reference, or an arithmetic expression on column references.

The BY clause causes the RMAX computation to restart at the row where the value of any of the BY columns differs from the previous row.

Example

This example shows a query that uses the RMAX function, along with example query results.

```
SELECT month, profit, RMAX(profit) from sales_subject_area
```

Result:

MONTH	PROFIT	RMAX(profit)
JAN	100.00	100.00
FEB	200.00	200.00
MAR	100.00	200.00
APRIL	100.00	200.00
MAY	300.00	300.00
JUNE	400.00	400.00
JULY	500.00	500.00
AUG	500.00	500.00
SEPT	500.00	500.00
OCT	300.00	500.00
NOV	200.00	500.00
DEC	100.00	500.00

RMIN

The RMIN function takes a set of records as input and shows the minimum value based on records encountered so far.

You must use a data type one that has an associated sort order.

Syntax

```
RMIN(expr)
```

Where:

expr is an expression of any data type. The data type must be one that has an associated sort order.

The RMIN function resets its values for each group in the query according to specific rules. See [Display Function Reset Behavior](#) for more information.

In AnswersOracle BI Answers, you can also use the following alternate syntax:

`RMIN(expression1 [BY expression2[, expression3[, ...]])`

Where:

`expression1`, `expression2`, `expression3` ... can be any column reference, or an arithmetic expression on column references.

The BY clause causes the RMIN computation to restart at the row where the value of any of the BY columns differs from the previous row.

Example

This example shows a query that uses the RMIN function, along with example query results.

```
select month, profit, RMIN(profit) from sales_subject_area
```

Result:

MONTH	PROFIT	RMIN(profit)
JAN	400.00	400.00
FEB	200.00	200.00
MAR	100.00	100.00
APRIL	100.00	100.00
MAY	300.00	100.00
JUNE	400.00	100.00
JULY	500.00	100.00
AUG	500.00	100.00
SEPT	500.00	100.00
OCT	300.00	100.00
NOV	200.00	100.00
DEC	100.00	100.00

Time Series Functions

Time series functions operate on time-oriented dimensions.

The time series functions calculate `AGO`, `TODATE`, and `PERIODROLLING` functions based on user supplied calendar tables, not on standard SQL date manipulation functions.

In Oracle BI Answers use time series functions on a particular dimension. You have to designate the dimension as a Time dimension and set one or more keys at one or more levels as chronological keys. See *About the Oracle BI Server Query Cache* in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

These functions let you use Expression Builder to call a logical function to perform time series calculations instead of aliasing physical tables and modeling logically.

To use time series functions on a particular dimension, you must designate the dimension as a Time dimension and set one or more keys at one or more levels as chronological keys. See *Modeling Time Series Data* in *Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Functions include:

- [AGO](#)
- [PERIODROLLING](#)

- [TODATE](#)

AGO

The AGO function calculates the aggregated value from the current time to a specified time period.

The AGO function is a time series aggregation function. AGO calculates the aggregated value from the current time grain, as determined by the measure grain (granular unit) and predicates, to a specified time period. For example, you can use the AGO function to produce sales for every month of the current quarter, and the corresponding quarter-ago sales.

Note:

The AGO function is not supported in Snowflake model.

The grain at which the time shift occurs is determined by the measure grain and predicates. For example, in the following query, the time shift occurs at the quarter level and goes back one year:

```
Select year, quarter, Ago(sales, year_level, 1)
```

However, in the following query, the time shift occurs at the day level, as determined by the associated level of the `day_of_month` attribute.

```
Select year, quarter, Ago(sales, year_level, 1)
Where day_of_month IN (1, 2)
```

When computing the time shift, Oracle BI Server steps through the time hierarchy. For example, if the time hierarchy is Year, then Month, then Day, then Oracle BI Server computes the time shift as follows:

1. Year is shifted by 1.
2. Month in year remains the same.
3. Day of month remains the same.

See [About the AGO Function Level](#) for more information about the level of the function.

If unsupported metrics are requested, NULL values are returned and a warning entry is written to the `nqquery.log` file when the logging level equals three or above.

You can nest multiple AGO functions if all the AGO functions have the same level argument. You can nest exactly one TODATE and multiple AGO functions if they each have the same level argument.

Syntax

```
AGO(expr, [time_level,] offset)
```

Where:

expr is an expression that references at least one measure column.

time_level is an optional argument that specifies the type of time period, such as quarter, month, or year.

In Answers, specify a presentation level from a presentation hierarchy for *time_level*.

In the Administration Tool, specify a logical level for *time_level*.

offset is an integer literal that represents the time shift amount.

Example

The following example returns last year's sales:

```
SELECT Year_ID, AGO(sales, year, 1)
```

About the AGO Function Level

It is recommended that you explicitly specify the level of the AGO function using the [*time_level*] argument.

If you do not explicitly specify the [*time_level*] argument, the default level is determined as follows:

- If the measure used in the expression is a level-based measure in the time dimension, as set in the Administration Tool, then that same level is considered the default AGO level.
- Otherwise, the grain of the measure used in the expression, as determined by the BY clause of the measure shown in the logical request, is the default Ago level.

For example, the result of the query:

```
SELECT year, AGO(sales, 1) WHERE quarter=1
```

is the same as:

```
SELECT year, AGO(sales, year_level, 1) WHERE quarter=1
```

You can see the default AGO level for a given query in the Logical Request section of the query log.

PERIODROLLING

The PERIODROLLING function computes the aggregate of a measure over the period starting *x* units of time and ending *y* units of time from the current time.

For example, you can use PERIODROLLING to compute sales for a period that starts at a specific quarter before and ends at a different quarter after the current quarter.

Because times series functions operate on members of time dimensions that are at or below the level of the function, one or more columns that uniquely identify members at or below the given level must be projected in the query. You can apply a filter to the query that specifies a single member at or below the given level. See [Determining the Level Used by the PERIODROLLING Function](#) for more information about the level of the function.

You cannot nest AGO and TODATE functions within a PERIODROLLING function. Also, you cannot nest PERIODROLLING, FIRST, FIRST_PERIOD, LAST, and LAST_PERIOD functions.

If you embed other aggregate functions such as RANK, TOPN, PERCENTILE, FILTER, or RSUM inside the PERIODROLLING function, the PERIODROLLING function is

pushed inward. For example, `PERIODROLLING(TOPN(measure))` is executed as `TOPN(PERIODROLLING(measure))`.

Syntax

```
PERIODROLLING(measure, x, y [,hierarchy])
```

Where:

measure is the name of a measure column.

x is an integer that specifies the offset from the current time. Precede the integer with a minus sign (-) to indicate an offset into the past.

y specifies the number of time units over which the function computes. To specify the current time, enter 0.

hierarchy is an optional argument that specifies the name of a hierarchy in a time dimension, such as *yr*, *mon*, *day*, that you want to use to compute the time window. This option is useful when there are multiple hierarchies in a time dimension, or when you want to distinguish between multiple time dimensions.

If you want to roll back or forward the maximum possible amount, use the keyword `UNBOUND`. For example, the function `PERIODROLLING (measure , -UNBOUND, 0)` sums over the period starting from the beginning of time until now.

You can combine `PERIODROLLING` and `AGGREGATE AT` functions to specify the level of the `PERIODROLLING` function explicitly. For example, if the query level is day but you want to find the sum of the previous and current months, use the following:

```
SELECT year, month, day, PERIODROLLING(AGGREGATE(sales AT month), -1)
```

Examples

```
SELECT Month_ID, PERIODROLLING(monthly_sales, -1, 1)
```

```
SELECT Month_ID, PERIODROLLING(monthly_sales, -UNBOUND, 2)
```

```
SELECT Month_ID, PERIODROLLING(monthly_sales, -UNBOUND, UNBOUND)
```

Determining the Level Used by the PERIODROLLING Function

The unit of time, offset, used in the `PERIODROLLING` function is called the level of the function.

This value is determined by the measure level of the measures in its first argument and the query level of the query to which the function belongs. You can set the measure level for the measure in the Administration Tool. If a measure level has been set for the measure used in the function, the measure level is used as the level of the function. The measure level is also called the *storage grain* of the function.

If a measure level has not been set in the Administration Tool, then the query level is used. The query level is also called the *query grain* of the function. In the following example, the query level is month, and the `PERIODROLLING` function computes the sum of the last, current, and next month for each city for the months of March and April:

```
SELECT year, month, country, city, PERIODROLLING(sales, -1, 1)
WHERE month in ('Mar', 'Apr') AND city = 'New York'
```

When there are multiple hierarchies in the time dimension, you must specify the *hierarchy* argument in the PERIODROLLING function. For example:

```
SELECT year, fiscal_year, month, PERIODROLLING(sales, -1, 1, "fiscal_time_hierarchy")
```

In this example, the level of the PERIODROLLING function is `fiscal_year`.

TODATE

The TODATE function is a time series aggregation function that combines a measure from the beginning of a specified time period to the currently displayed time. For example, this function can calculate Year to Date sales.

Time series functions operate on members of time dimensions which are at or below the level specified in the function. You must set one or more columns that uniquely identify members at or below the given level in the query. You can apply a filter to the query that specifies a single member at or below the given level.

If unsupported metrics are requested, NULL values are returned and a warning entry is written to the `nqquery.log` file when the logging level equals three or above.

A TODATE function may not be nested within another TODATE function. You can nest exactly one TODATE and multiple AGO functions if they each have the same level argument.

TODATE is different from the TO_DATE SQL function supported by some databases. Do not use TO_DATE to change to a DATE data type. Instead, use the CAST function. See [CAST](#) for more information.

Syntax

```
TODATE(expr, time_level)
```

Where:

expr is an expression that references at least one measure column.

time_level is the type of time period, such as quarter, month, or year.

Example

The following example returns the year-to-month sales:

```
SELECT Year_ID, Month_ID, TODATE(sales, year)
```

Reporting Functions

Reporting functions are computed after all predicates have been evaluated and are useful when computing subtotals of values visible in the report.

Reporting functions are only allowed in the project list (SELECT clause).

These functions can only be used in Logical SQL and cannot be used in expressions inside the Oracle BI repository.

Functions include:

- [REPORT_AGGREGATE](#)
- [REPORT_AVG](#)
- [REPORT_COUNT](#)
- [REPORT_MAX](#)
- [REPORT_MIN](#)
- [REPORT_SUM](#)

REPORT_AGGREGATE

The `REPORT_AGGREGATE` function is a generalization of other reporting functions like `REPORT_SUM`, `REPORT_MIN`, and similar functions.

It is semantically equivalent to those functions when applied on a measure with the corresponding aggregation rule.

For example, if measure `M` is defined with the aggregation rule `SUM`, then:

```
REPORT_AGGREGATE(M) := REPORT_SUM(M)
```

However, if the aggregation rule is `MAX`, then:

```
REPORT_AGGREGATE(M) := REPORT_MAX(M)
```

`REPORT_AGGREGATE` is only allowed in the project list (`SELECT` clause) and is computed after all expressions in the `WHERE` clause have been evaluated, with one exception: it is computed before the aggregate predicates when applied to a non-linear measure, such as `COUNTDISTINCT`.

Note:

If you do not want `REPORT_AGGREGATE` computed before the aggregate predicates on non-linear measures, re-write your queries so that they do not have any summaries in the `WHERE` clause. For example, you could rewrite the following query:

```
SELECT Year, DistinctUnitsSold,
REPORT_AGGREGATE(DistinctUnitsSold BY )
WHERE Sales > 1000
```

as:

```
SELECT Year, DistinctProductsSold, REPORT_AGGREGATE(DistinctUnitsSold BY )
WHERE Year IN (
SELECT Year
WHERE sales > 1000
)
```

You can declare the non-linear measure as a derived column instead of a measure. For example:

```
COUNT(DISTINCT UnitId)
```

Because `REPORT_AGGREGATE` is computed after evaluating all predicates, the function is useful when computing subtotals of values visible in the report.

When `REPORT_AGGREGATE` is applied to an expression `E` of measures (`M1`, `M2`, and so on) and scalar attributes (`S1`, `S2`, and so on), it is distributed to the component measures, as follows:

```
REPORT_AGGREGATE(E(M1, M2, ..., S1, S2, ...)) := E(REPORT_AGGREGATE(M1),
REPORT_AGGREGATE(M2), ..., S1, S2, ...)
```

For example:

```
REPORT_AGGREGATE((sales + 10) / number_of_employees) := (REPORT_AGGREGATE(sales) +
10) / REPORT_AGGREGATE(number_of_employees)
```

REPORT_AVG

The `REPORT_AVG` function computes the average (mean) value of the `argument_expression` partitioned by the set of `BY` column expressions.

An empty `BY` clause specifies a single (grand total) partition.

Define the columns in the `BY` clause as a subset of the (implicit) `GROUP BY` columns of the `argument_expression`.

`REPORT_AVG` is only allowed in the project list, `SELECT` clause, and is computed after all expressions in the `WHERE` clause.

Because `REPORT_AVG` is computed after all other predicates, it is suitable for computing report subtotal values.

While `REPORT_AVG` always uses `AVG` as the aggregation rule, `REPORT_AGGREGATE` tries to automatically determine the aggregation rule, based on its argument.

Oracle recommends using `REPORT_AGGREGATE` by default. However, when the default behavior of `REPORT_AGGREGATE` does not produce the desired result, you can use an explicit aggregation rule like `REPORT_AVG`.

Syntax

```
REPORT_AVG(argument_expression BY column_expression,
column_expression, ...)
```

Example

```
SELECT month, year, sales, REPORT_AVG(sales BY year) AS yearly_total
WHERE month LIKE 'J%' AND sales > 10
```

Result:

The table shows the results for the `REPORT_SUM` example.

Month	Year	Sales	Yearly_Total
June	2011	20	25
July	2011	30	25
January	2012	40	45
June	2012	50	45

REPORT_COUNT

The REPORT_COUNT function calculates the number of rows having a nonnull value for the *argument_expression* partitioned by the set of BY column expressions.

An empty BY clause specifies a single (grand total) partition.

Columns in the BY clause must be a subset of the (implicit) GROUP BY columns of the *argument_expression*.

REPORT_COUNT is only allowed in the project list (SELECT clause) and is computed after all expressions in the WHERE clause.

Because REPORT_COUNT is computed after all other predicates, it is suitable for computing report subtotal values.

While REPORT_COUNT always uses COUNT as the aggregation rule, REPORT_AGGREGATE tries to automatically determine the aggregation rules, based on its argument.

It is recommended to use REPORT_AGGREGATE by default. However, when the default behavior of REPORT_AGGREGATE does not produce the desired result, you can use an explicit aggregation rule like REPORT_COUNT.

Syntax

```
REPORT_COUNT(argument_expression BY column_expression, column_expression, ...)
```

Example

```
SELECT month, year, sales, REPORT_COUNT(sales BY year) AS yearly_total  
WHERE month LIKE 'J%' AND sales > 10
```

Result:

Month	Year	Sales	Yearly_Total
June	2011	20	2
July	2011	30	2
January	2012	40	2
June	2012	50	2

REPORT_MAX

The REPORT_MAX function calculates the maximum value (highest numeric value) of the rows satisfying the numeric *argument_expression* partitioned by the set of BY column expressions.

An empty BY clause specifies a single (grand total) partition.

Columns in the BY clause must be a subset of the (implicit) GROUP BY columns of the *argument_expression*.

REPORT_MAX is only allowed in the project list (SELECT clause) and is computed after all expressions in the WHERE clause.

Because `REPORT_MAX` is computed after all other predicates, it is suitable for computing report subtotal values.

While `REPORT_MAX` always uses `MAX` as the aggregation rule, `REPORT_AGGREGATE` tries to automatically determine the aggregation rules, based on its argument.

It is recommended to use `REPORT_AGGREGATE` by default. However, when the default behavior of `REPORT_AGGREGATE` does not produce the desired result, You can use an explicit aggregation rule like `REPORT_MAX`.

Syntax

```
REPORT_MAX(argument_expression BY column_expression, column_expression, ...)
```

Example

```
SELECT month, year, sales, REPORT_MAX(sales BY year) AS yearly_total  
WHERE month LIKE 'J%' AND sales > 10
```

Result:

Month	Year	Sales	Yearly_Total
June	2011	20	30
July	2011	30	30
January	2012	40	50
June	2012	50	50

REPORT_MIN

The `REPORT_MIN` function calculates the minimum value (lowest numeric value) of the rows satisfying the *argument_expression* partitioned by the set of `BY` column expressions.

An empty `BY` clause specifies a single (grand total) partition.

Columns in the `BY` clause must be a subset of the (implicit) `GROUP BY` columns of the *argument_expression*.

`REPORT_MIN` is only allowed in the project list (`SELECT` clause) and is computed after all expressions in the `WHERE` clause.

Because `REPORT_MIN` is computed after all other predicates, it is suitable for computing report subtotal values.

While `REPORT_MIN` always uses `MIN` as the aggregation rule, `REPORT_AGGREGATE` tries to automatically determine the aggregation rules, based on its argument.

It is recommended to use `REPORT_AGGREGATE` by default. However, when the default behavior of `REPORT_AGGREGATE` does not produce the desired result, you can use an explicit aggregation rule like `REPORT_MIN`.

Syntax

```
REPORT_MIN(argument_expression BY column_expression, column_expression, ...)
```

Example

```
SELECT month, year, sales, REPORT_MIN(sales BY year) AS yearly_total
WHERE month LIKE 'J%' AND sales > 10
```

Result:

Month	Year	Sales	Yearly_Total
June	2011	20	20
July	2011	30	20
January	2012	40	40
June	2012	50	40

REPORT_SUM

The REPORT_SUM function computes the sum of the argument_expression partitioned by the set of BY column expressions.

BY clause must be a subset of the (implicit) GROUP BY columns of the argument_expression.

REPORT_SUM is only allowed in the project list (SELECT clause) and is computed after all expressions in the WHERE clause.

Because REPORT_SUM is computed after all other predicates, it is suitable for computing report subtotal values.

While REPORT_SUM always uses SUM as the aggregation rule, REPORT_AGGREGATE tries to automatically determine the aggregation rule, based on its argument.

It is recommended to use REPORT_AGGREGATE by default. However, when the default behavior of REPORT_AGGREGATE does not produce the desired result, you can use an explicit aggregation rule like REPORT_SUM. For example, use REPORT_SUM to compute totals or subtotals as a simple visual sum of the values displayed in a report rather than using the same aggregation rule as the base measure.

Syntax

```
REPORT_SUM(argument_expression BY column_expression, column_expression, ...)
```

Example

```
SELECT month, year, sales, REPORT_SUM(sales BY year) AS yearly_total
WHERE month LIKE 'J%' AND sales > 10
```

Result:

Month	Year	Sales	Yearly_Total
June	2011	20	50
July	2011	30	50

Month	Year	Sales	Yearly_Total
January	2012	40	90
June	2012	50	90

String Functions

String functions perform various character manipulations, and they operate on character strings.

Functions include:

- [ASCII](#)
- [BIT_LENGTH](#)
- [CHAR](#)
- [CHAR_LENGTH](#)
- [CONCAT](#)
- [INSERT](#)
- [LEFT](#)
- [LENGTH](#)
- [LOCATE](#)
- [LOWER](#)
- [OCTET_LENGTH](#)
- [POSITION](#)
- [REPEAT](#)
- [REPLACE](#)
- [RIGHT](#)
- [SPACE](#)
- [SUBSTRING](#)
- [TRIMBOTH](#)
- [TRIMLEADING](#)
- [TRIMTRAILING](#)
- [UPPER](#)

ASCII

The ASCII function converts a single character string to its corresponding ASCII code.

This function converts a single character string to its corresponding ASCII code, between 0 and 255. If the character expression evaluates to multiple characters, the ASCII code corresponding to the first character in the expression is returned.

Syntax

`ASCII(strExpr)`

Where:

strExpr is any expression that evaluates to a character string.

BIT_LENGTH

The BIT_LENGTH function returns the length, in bits, of a specified string.

Each Unicode character is 2 bytes in length (equal to 16 bits).

Syntax

`BIT_LENGTH(strExpr)`

Where:

strExpr is any expression that evaluates to character string.

CHAR

The CHAR function converts a numeric value between 0 and 255 to the character value corresponding to the ASCII code.

Syntax

`CHAR(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value between 0 and 255.

CHAR_LENGTH

The CHAR_LENGTH function returns the length, in number of characters, of a specified string. Leading and trailing blanks are not counted in the length of the string.

Syntax

`CHAR_LENGTH(strExpr)`

Where:

strExpr is any expression that evaluates to a character string.

CONCAT

There are two forms of the CONCAT function.

The first form concatenates two character strings. The second form uses the character string concatenation character to concatenate more than two character strings.

Syntax for Form 1 (To Concatenate Two Strings)

```
CONCAT(strExpr1, strExpr2)
```

Where:

strExprs are expressions that evaluate to character strings, separated by commas.

Example

This example request returns the results shown.

```
SELECT DISTINCT CONCAT('abc', 'def') FROM employee  
CONCAT('abc', 'def')
```

Result:

```
abcdef
```

Syntax for Form 2 (To Concatenate More Than Two Strings)

```
CONCAT(strExpr1, strExpr2 || strExpr3)
```

Where:

strExprs are expressions that evaluate to character strings, separated by commas and the character string concatenation operator || (double vertical bars). First, *strExpr2* is concatenated with *strExpr3* to produce an intermediate string, then both *strExpr1* and the intermediate string are concatenated by the CONCAT function to produce the final string.

Example

This example request returns the results shown.

```
SELECT DISTINCT CONCAT('abc','def' || 'ghi') FROM employee
```

Result:

```
abcdefghi
```

INSERT

The INSERT function interjects a specified character string into a specified location in another character string.

Syntax

```
INSERT(strExpr1, integer1, integer2, strExpr2)
```

Where:

strExpr1 is any expression that evaluates to a character string. Identifies the target character string.

integer1 is any positive integer that represents the number of characters from the beginning of the target string where the second string is to be inserted.

integer2 is any positive integer that represents the number of characters in the target string to be replaced by the second string.

strExpr2 is any expression that evaluates to a character string. Identifies the character string to be inserted into the target string.

Example

In the first string, starting at the second position (occupied by the number 2), three characters (the numbers 2, 3, and 4) are replaced by the string `abcd`.

```
SELECT INSERT('123456', 2, 3, 'abcd') FROM table
```

Result:

```
1abcd56
 1abcd56
  ...
```

LEFT

The `LEFT` function returns a specified number of characters from the left of a string.

Syntax

```
LEFT(strExpr, integer)
```

Where:

strExpr is any expression that evaluates to a character string.

integer is any positive integer that represents the number of characters from the left of the string to return.

Example

This example returns the three leftmost characters from the character string `123456`:

```
SELECT LEFT('123456', 3) FROM table
```

Result:

```
123
123
  ...
```

LENGTH

The `LENGTH` function returns the measurement, in number of characters, of a specified string.

The length is returned excluding any trailing blank characters.

Syntax

```
LENGTH(strExpr)
```

Where:

strExpr is any expression that evaluates to a character string.

LOCATE

The LOCATE function returns the numeric position of a character string in another character string.

If the character string is not found in the string being searched, the function returns a value of 0.

If you want to specify a starting position to begin the search, include the integer argument. The numeric position to return is determined by counting the first character in the string as occupying position 1, regardless of the value of the integer argument.

Syntax

```
LOCATE(strExpr1, strExpr2 [, integer])
```

Where:

strExpr1 is any expression that evaluates to a character string. Identifies the string for which to search.

strExpr2 is any expression that evaluates to a character string. Identifies the string to be searched.

integer is any positive (nonzero) integer that represents the starting position to begin to look for the character string. The integer argument is optional.

Examples

This example returns 4 as the numeric position of the letter d in the character string abcdef:

```
LOCATE('d', 'abcdef')
```

This example returns 0, because the letter g is not found within the string being searched.

```
LOCATE('g', 'abcdef')
```

This example returns 4 as the numeric position of the letter d in the character string abcdef. The search begins with the letter c, the third character in the string. The numeric position to return is determined by counting the letter 'a' as occupying position 1.

```
LOCATE('d' 'abcdef', 3)
```

This example returns 0, because the letter b occurs in the string before the starting position to begin the search.

```
LOCATE('b' 'abcdef', 3)
```

LOWER

The LOWER function converts a character string to lowercase.

Syntax

```
LOWER(strExpr)
```

Where:

strExpr is any expression that evaluates to a character string.

OCTET_LENGTH

The OCTET_LENGTH function returns the number of bits, in base 8 units (number of bytes), of a specified string.

Syntax

```
OCTET_LENGTH(strExpr)
```

Where:

strExpr is any expression that evaluates to a character string.

POSITION

The POSITION function returns the numeric position of *strExpr1* in a character expression.

If *strExpr1* is not found, the function returns 0. See also [LOCATE](#) for related information.

Syntax

```
POSITION(strExpr1 IN strExpr2)
```

Where:

strExpr1 is any expression that evaluates to a character string. Identifies the string to search for in the target string.

strExpr2 is any expression that evaluates to a character string. Identifies the target string to be searched.

Examples

This example returns 4 as the position of the letter d in the character string abcdef:

```
POSITION('d', 'abcdef')
```

This example returns 0 as the position of the number 9 in the character string 123456, because the number 9 is not found.

```
POSITION('9', '123456')
```

REPEAT

The REPEAT function reruns a specified expression *n* times.

Syntax

```
REPEAT(strExpr, integer)
```

Where:

strExpr is any expression that evaluates to a character string.

integer is any positive integer that represents the number of times to repeat the character string.

Example

This example repeats abc four times:

```
REPEAT('abc', 4)
```

REPLACE

The REPLACE function changes one or more characters from a specified character expression with one or more other characters.

Syntax

```
REPLACE(strExpr1, strExpr2, strExpr3)
```

Where:

strExpr1 is any expression that evaluates to a character string. This is the string in which characters are to be replaced.

strExpr2 is any expression that evaluates to a character string. This second string identifies the characters from the first string that are to be replaced.

strExpr3 is any expression that evaluates to a character string. This third string specifies the characters to substitute into the first string.

Example

In the character string abcd1234, the characters 123 are replaced by the character string ZZ:

```
Replace('abcd1234', '123', 'zz')
```

Result:

```
abcdzz4
```

RIGHT

The RIGHT function returns a specified number of characters from the right of a string.

Syntax

```
RIGHT(strExpr, integer)
```

Where:

strExpr is any expression that evaluates to a character string.

integer is any positive integer that represents the number of characters from the right of the string to return.

Example

This example returns the three rightmost characters from the character string 123456:

```
SELECT right('123456', 3) FROM table
```

Result:

```
456
```

SPACE

The SPACE function inserts blank spaces.

Syntax

```
SPACE(integer)
```

Where:

integer is any positive integer that indicates the number of spaces to insert.

SUBSTRING

The SUBSTRING function creates a new string starting from a fixed number of characters into the original string.

Syntax

```
SUBSTRING(strExpr FROM starting_position)
```

Where:

strExpr is any expression that evaluates to a character string.

starting_position is any positive integer that represents the number of characters from the start of the left side of the string where the result is to begin.

TRIMBOTH

The TRIMBOTH function strips specified leading and trailing characters from a character string.

Syntax

TRIM(BOTH *character* FROM *strExpr*)

Where:

character is any single character. If you omit this specification (and the required single quotes), a blank character is used as the default.

strExpr is any expression that evaluates to a character string.

TRIMLEADING

The TRIMLEADING function strips specified leading characters from a character string.

Syntax

TRIM(LEADING *character* FROM *strExpr*)

Where:

character is any single character. If you omit this specification and the required single quotes, a blank character is used as the default.

strExpr is any expression that evaluates to a character string.

TRIMTRAILING

The TRIMTRAILING function strips specified trailing characters from a character string.

Syntax

TRIM(TRAILING *character* FROM *strExpr*)

Where:

character is any single character. If you omit this specification and the required single quotes, a blank character is used as the default.

strExpr is any expression that evaluates to a character string.

UPPER

The UPPER function converts a character string to uppercase.

Syntax

UPPER(*strExpr*)

Where:

strExpr is any expression that evaluates to a character string.

Math Functions

These math functions perform mathematical operations.

Functions include:

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- EXTRACTBIT
- FLOOR
- LOG
- LOG10
- MOD
- PI
- POWER
- RADIANS
- RAND
- RANDFROMSEED
- ROUND
- SIGN
- SIN
- SQRT
- TAN
- TRUNCATE

ABS

The ABS function calculates the absolute value of a numeric expression.

Syntax

`ABS(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

ACOS

The ACOS function calculates the arc cosine of a numeric expression.

Syntax

`ACOS(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

ASIN

The ASIN function calculates the arc sine of a numeric expression.

Syntax

`ASIN(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

ATAN

The ATAN function calculates the arc tangent of a numeric expression.

Syntax

`ATAN(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

ATAN2

The ATAN2 function calculates the arc tangent of y/x .

This function calculates the arc tangent of y/x , where y is the first numeric expression and x is the second numeric expression.

Syntax

`ATAN2(numExpr1, numExpr2)`

Where:

numExpr is any expression that evaluates to a numeric value.

CEILING

The CEILING function rounds a noninteger numeric expression to the next highest integer.

If the numeric expression evaluates to an integer, the CEILING function returns that integer.

Syntax

`CEILING(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

COS

The COS function calculates the cosine of a numeric expression.

Syntax

`COS(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

COT

The COT function calculates the cotangent of a numeric expression.

Syntax

`COT(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

DEGREES

The DEGREES function converts an expression from radians to degrees.

Syntax

`DEGREES(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

EXP

The EXP function sends the value to the power specified.

Syntax

EXP(*numExpr*)

Where:

numExpr is any expression that evaluates to a numeric value.

EXTRACTBIT

The EXTRACTBIT function retrieves a bit at a particular position in an integer.

It returns an integer of either 0 or 1 corresponding to the position of the bit. The primary use case for this function is to extract 'cell status' in the Hyperion Financial Management cube source. The EXTRACTBIT function cannot be pushed into any database, and is always internally executed in the Oracle BI Server.

Syntax

```
Int ExtractBit(Arg1, Arg2)
```

Where:

Arg1 is an expression of the following types: INT, SMALLINT, UNIT, SMALLUNIT, TINYINT, TINYUNIT. If *Arg1* is of double type, it is necessary to cast the column to an INT first.

Arg2 is an expression of type integer. The value should range from 1 to *length_of_Arg1*. 1 retrieves the Least Significant Bit. If the *Arg2* is beyond the length of the integer, then 0 is returned. An error message is triggered when the *Arg2* is less than 1.

FLOOR

The FLOOR function rounds a non-integer numeric expression to the next lowest integer.

If the numeric expression evaluates to an integer, the FLOOR function returns that integer.

Syntax

```
FLOOR(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

LOG

The LOG function calculates the natural logarithm of an expression.

Syntax

```
LOG(numExpr)
```

Where:

numExpr is any expression that evaluates to a numeric value.

LOG10

The LOG10 function calculates the base 10 logarithm of an expression.

Syntax

`LOG10(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

MOD

The MOD function divides the first numeric expression by the second numeric expression and returns the remainder portion of the quotient.

Syntax

`MOD(numExpr1, numExpr2)`

Where:

numExpr is any expression that evaluates to a numeric value.

Examples

This example request returns a value of 0:

`MOD(9, 3)`

This example request returns a value of 1:

`MOD(10, 3)`

PI

The PI function returns the constant value of pi, the circumference of a circle divided by its diameter.

Syntax

`PI()`

POWER

The POWER function takes the first numeric expression and raises it to the power specified in the second numeric expression.

Syntax

`POWER(numExpr1, numExpr2)`

Where:

numExpr1 is any expression that evaluates to a numeric value.

RADIANS

The RADIANS function converts an expression from degrees to radians.

Syntax

`RADIANS(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

RAND

The RAND function returns a pseudo-random number between 0 and 1.

Syntax

`RAND()`

RANDFROMSEED

The RANDFROMSEED function returns a pseudo-random number based on a seed value.

For a given seed value, the same set of random numbers are generated.

Syntax

`RAND(numExpr)`

Where:

numExpr is any expression that evaluates to a numeric value.

ROUND

The ROUND function rounds a numeric expression to *n* digits of precision.

Syntax

`ROUND(numExpr, integer)`

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer that represents the number of digits of precision.

Example

This example returns 2.17 as the result.

`ROUND(2.166000, 2)`

SIGN

The SIGN function returns a value that depends on the numeric expression argument that it evaluates.

The SIGN function returns the following:

- A value of 1 if the numeric expression argument evaluates to a positive number.
- A value of -1 if the numeric expression argument evaluates to a negative number.
- 0 (zero) if the numeric expression argument evaluates to zero.

Syntax

SIGN(numExpr)

Where:

numExpr is any expression that evaluates to a numeric value.

SIN

The SIN function calculates the sine of a numeric expression.

Syntax

SIN(numExpr)

Where:

numExpr is any expression that evaluates to a numeric value.

SQRT

The SQRT function calculates the square root of the numeric expression argument.

The numeric expression must evaluate to a nonnegative number.

Syntax

SQRT(numExpr)

Where:

numExpr is any expression that evaluates to a nonnegative numeric value.

TAN

The TAN function calculates the tangent of a numeric expression.

Syntax

TAN(numExpr)

Where:

numExpr is any expression that evaluates to a numeric value.

TRUNCATE

The TRUNCATE function trims a decimal number to return a specified number of places from the decimal point.

Syntax

```
TRUNCATE(numExpr, integer)
```

Where:

numExpr is any expression that evaluates to a numeric value.

integer is any positive integer that represents the number of characters to the right of the decimal place to return.

Examples

This example returns 45.12:

```
TRUNCATE(45.12345, 2)
```

This example returns 25.12:

```
TRUNCATE(25.126, 2)
```

Calendar Date/Time Functions

The calendar date/time functions manipulate data from DATE and DATETIME data types based on a calendar year.

You must select these functions with another column; they cannot be selected alone.

Month and day names are returned in mixed case by default. If you have existing expressions that assume month/day names are returned in uppercase, you must either adjust the expressions, or set the parameters `USE_UPPERCASE_MONTH_NAMES` and `USE_UPPERCASE_DAY_NAMES` to YES in `NQSConfig.INI`. See `NQSConfig.INI` File Configuration Settings in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

You can use functions such as `MONTHNAME` to format date columns, but the locale is fixed by the data source or the Oracle BI Server's `NQSConfig.INI` file. You cannot specify a locale for the Logical SQL date formatting functions because the locales are based on the ODBC standard that does not provide for a locale argument.

To format date columns according to the user's selected locale, Oracle suggests that when the locale is fixed, the content designer specifies a standard date format or a custom date format on an analysis' date column. See Custom Format Strings for Date and Time Fields in *User's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Functions include:

- [CURRENT_DATE](#)
- [CURRENT_TIME](#)
- [CURRENT_TIMESTAMP](#)

- DAY_OF_QUARTER
- DAYNAME
- DAYOFMONTH
- DAYOFWEEK
- DAYOFYEAR
- HOUR
- MINUTE
- MONTH
- MONTH_OF_QUARTER
- MONTHNAME
- NOW
- QUARTER_OF_YEAR
- SECOND
- TIMESTAMPADD
- TIMESTAMPDIFF
- WEEK_OF_QUARTER
- WEEK_OF_YEAR
- YEAR

CURRENT_DATE

The CURRENT_DATE function returns the current date.

The date is determined by the system in which the Oracle BI Server is running.

Syntax

CURRENT_DATE

CURRENT_TIME

The CURRENT_TIME function returns the current time.

The time is determined by the system in which the Oracle BI Server is running.

Syntax

CURRENT_TIME(*integer*)

Where:

integer is any integer representing the number of digits of precision with which to display the fractional second. The argument is optional; the function returns the default precision when no argument is specified.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns the current date/timestamp.

The timestamp is determined by the system in which the Oracle BI Server is running.

Syntax

```
CURRENT_TIMESTAMP(integer)
```

Where:

integer is any integer representing the number of digits of precision with which to display the fractional second. The argument is optional; the function returns the default precision when no argument is specified.

DAY_OF_QUARTER

The DAY_OF_QUARTER function returns a number, between 1 and 92, corresponding to the day of the quarter for the specified date.

Syntax

```
DAY_OF_QUARTER(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

DAYNAME

The DAYNAME function returns the name of the day of the week for a specified date.

Syntax

```
DAYNAME(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

DAYOFMONTH

The DAYOFMONTH function returns the number corresponding to the day of the month for a specified date.

Syntax

```
DAYOFMONTH(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

DAYOFWEEK

The DAYOFWEEK function returns a number between 1 and 7 corresponding to the day of the week, Sunday through Saturday, for a specified date.

For example, the DAYOFWEEK function returns a number 1 to correspond to Sunday, and the number 7 to correspond to Saturday.

Syntax

`DAYOFWEEK(dateExpr)`

Where:

dateExpr is any expression that evaluates to a date.

DAYOFYEAR

The DAYOFYEAR function returns a number, between 1 and 366, corresponding to the day of the year for a specified date.

Syntax

`DAYOFYEAR(dateExpr)`

Where:

dateExpr is any expression that evaluates to a date.

HOUR

The HOUR function returns a number (between 0 and 23) corresponding to the hour for a specified time.

For example, 0 corresponds to 12 a.m. and 23 corresponds to 11 p.m.

Syntax

`HOUR(timeExpr)`

Where:

timeExpr is any expression that evaluates to a time.

MINUTE

The MINUTE function returns a number (between 0 and 59) corresponding to the minute for a specified time.

Syntax

`MINUTE(timeExpr)`

Where:

timeExpr is any expression that evaluates to a time.

MONTH

The MONTH function returns the number (between 1 and 12) corresponding to the month for a specified date.

Syntax

```
MONTH(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

MONTH_OF_QUARTER

The MONTH_OF_QUARTER function returns the number (between 1 and 3) corresponding to the month in the quarter for a specified date.

Syntax

```
MONTH_OF_QUARTER(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

MONTHNAME

The MONTHNAME function returns the name of the month for a specified date.

Syntax

```
MONTHNAME(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

NOW

The NOW function returns the current timestamp.

The NOW function is equivalent to the CURRENT_TIMESTAMP function.

Syntax

```
NOW()
```

QUARTER_OF_YEAR

The QUARTER_OF_YEAR function returns the number (between 1 and 4) corresponding to the quarter of the year for a specified date.

Syntax

`QUARTER_OF_YEAR(dateExpr)`

Where:

dateExpr is any expression that evaluates to a date.

SECOND

The SECOND function returns the number (between 0 and 59) corresponding to the seconds for a specified time.

Syntax

`SECOND(timeExpr)`

Where:

timeExpr is any expression that evaluates to a time.

TIMESTAMPADD

The TIMESTAMPADD function adds a specified number of intervals to a specified timestamp, and returns a single timestamp.

In the simplest scenario, this function adds the specified integer value to the appropriate component of the timestamp, based on the interval. Adding a week translates to adding seven days, and adding a quarter translates to adding three months. A negative integer value results in a subtraction such as going back in time.

An overflow of the specified component such as more than 60 seconds, 24 hours, and 12 months, necessitates adding an appropriate amount to the next component. For example, when adding to the day component of a timestamp, this function considers overflow and takes into account the number of days in a particular month, including leap years when February has 29 days.

When adding to the month component of a timestamp, this function verifies that the resulting timestamp has enough days for the day component. For example, adding 1 month to 2000-05-31 does not result in 2000-06-31 because June does not have 31 days. This function reduces the day component to the last day of the month, 2000-06-30 in this example.

A similar issue arises when adding to the year component of a timestamp having a month component of February and a day component of 29, that is, last day of February in a leap year. If the resulting timestamp does not fall on a leap year, the function reduces the day component to 28.

These actions conform to the behavior of Microsoft SQL Server and the native OCI interface for Oracle Database.

Syntax

`TIMESTAMPADD(interval, intExpr, timestamp)`

Where:

interval is the specified interval. Valid values are:

- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

intExpr is any expression that evaluates to an integer value.

timestamp is any valid timestamp. This value is used as the base in the calculation.

A null integer expression or a null timestamp passed to this function results in a null return value.

Examples

The following query asks for the resulting timestamp when 3 days are added to 2000-02-27 14:30:00. Since February, 2000 is a leap year, the query returns a single timestamp of 2000-03-01 14:30:00.

```
SELECT TIMESTAMPADD(SQL_TSI_DAY, 3, TIMESTAMP'2000-02-27 14:30:00')
FROM Employee WHERE employeeid = 2;
```

The following query asks for the resulting timestamp when 7 months are added to 1999-07-31 0:0:0. The query returns a single timestamp of 2000-02-29 00:00:00. Notice the reduction of day component to 29 because of the shorter month of February.

```
SELECT TIMESTAMPADD(SQL_TSI_MONTH, 7, TIMESTAMP'1999-07-31 00:00:00')
FROM Employee WHERE employeeid = 2;
```

The following query asks for the resulting timestamp when 25 minutes are added to 2000-07-31 23:35:00. The query returns a single timestamp of 2000-08-01 00:00:00. Notice the propagation of overflow through the month component.

```
SELECT TIMESTAMPADD(SQL_TSI_MINUTE, 25, TIMESTAMP'2000-07-31 23:35:00')
FROM Employee WHERE employeeid = 2;
```

Caution:

The `TIMESTAMPADD` function is turned on by default for Microsoft SQL Server, ODBC, IBM DB2, and Oracle databases. Because DB2 and Oracle semantics do not fully support this function, the answers from this function might not match with what the Oracle BI Server computes.

TIMESTAMPDIFF

The `TIMESTAMPDIFF` function returns the total number of specified intervals between two timestamps.

This function first determines the timestamp component that corresponds to the specified interval parameter, and then looks at the higher order components of both timestamps to calculate the total number of intervals for each timestamp. For example, if the specified interval corresponds to the month component, the function calculates the total number of months for each timestamp by adding the month component and twelve times the year component. Then the function subtracts the first timestamp's total number of intervals from the second timestamp's total number of intervals.

Note:

This section describes the `TIMESTAMPDIFF` behavior when the function is calculated in the Oracle BI Server. If this function is calculated in the data source, then the result might be different from the behavior described in this section. If the `TIMESTAMPDIFF` function result is different from the desired result, then you can disable `TIMESTAMP_DIFF_SUPPORTED` in the Features tab for the database object in the Administration Tool to ensure that the function is calculated in the Oracle BI Server. However, making this change might adversely affect performance.

The `TIMESTAMPDIFF` function rounds up to the next integer whenever fractional intervals represent a crossing of an interval boundary. For example, the difference in years between 1999-12-31 and 2000-01-01 is one year because the fractional year represents a crossing from one year to the next (such as 1999 to 2000). By contrast, the difference between 1999-01-01 and 1999-12-31 is zero years because the fractional interval falls entirely within a particular year (that is, 1999). Microsoft SQL Server exhibits the same rounding behavior, but IBM DB2 does not; it always rounds down.

When calculating the difference in weeks, the function calculates the difference in days and divides by seven before rounding. Additionally, the function takes into account how the parameter `FIRST_DAY_OF_THE_WEEK` has been configured in the `NQSConfig.INI` file. For example, with Sunday as the start of the week, the difference in weeks between 2000-07-06 (a Thursday) and 2000-07-10 (the following Monday) results in a value of 1 week. With Tuesday as the start of the week, however, the function would return zero weeks since the fractional interval falls entirely within a particular week. When calculating the difference in quarters, the function calculates the difference in months and divides by three before rounding.

The Oracle BI Server pushes down the `TIMESTAMPADD` and `TIMESTAMPDIFF` functions to Microsoft SQL Server, Oracle Database, IBM DB2, and ODBC databases by default.

Syntax

```
TIMESTAMPDIFF(interval, timestamp1, timestamp2)
```

Where:

interval is the specified interval. Valid values are:

- `SQL_TSI_SECOND`

- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_YEAR

timestamp1 and *timestamp2* are any valid timestamps.

A null timestamp parameter passed to this function results in a null return value.

Example

The following example query asks for a difference in days between timestamps 1998-07-31 23:35:00 and 2000-04-01 14:24:00. It returns a value of 610. Notice that the leap year in 2000 results in an additional day.

```
SELECT TIMESTAMPDIFF
(SQL_TSI_DAY, TIMESTAMP'1998-07-31 23:35:00',TIMESTAMP'2000-04-01 14:24:00')
FROM Employee WHERE employeeid = 2;
```

Caution:

The TIMESTAMPDIFF function is turned on by default for Microsoft SQL Server, ODBC, IBM DB2, and Oracle databases. Because DB2 and Oracle semantics do not fully support this function, the answers from this function might not match with what the Oracle BI Server computes.

WEEK_OF_QUARTER

The WEEK_OF_QUARTER function returns a number (between 1 and 13) corresponding to the week of the quarter for the specified date.

Syntax

```
WEEK_OF_QUARTER(dateExpr)
```

Where:

dateExpr is any expression that evaluates to a date.

WEEK_OF_YEAR

The WEEK_OF_YEAR function returns a number (between 1 and 53) corresponding to the week of the year for the specified date.

Syntax

`WEEK_OF_YEAR(dateExpr)`

Where:

dateExpr is any expression that evaluates to a date.

YEAR

The YEAR function returns the year for the specified date.

Syntax

`YEAR(dateExpr)`

Where:

dateExpr is any expression that evaluates to a date.

Conversion Functions

The conversion functions transform a value from one form to another.

You can use the VALUEOF function in a filter to reference the value of an Oracle BI EE system variable. Functions include:

- [CAST](#)
- [CHOOSE](#)
- [IFNULL](#)
- [INDEXCOL](#)
- [TO_DATETIME](#)
- [VALUEOF](#)
- [VARIABLE_REPLACE](#)

CAST

The CAST function changes the data type of an expression or a null literal to another data type.

For example, you can cast a customer_name, a data type of Char or Varchar, or birthdate, a datetime literal. You can change the value in the following supported data types:

CHARACTER, VARCHAR, INTEGER, FLOAT, SMALLINT, DOUBLE PRECISION, DATE, TIME, TIMESTAMP, BIT, BIT VARYING

Depending on the source data type, some destination types are not supported. For example, if the source data type is a BIT string, the destination data type must be a character string or another BIT string.

Use CAST to change to a DATE data type.

The following describes unique characteristics of the CHAR and VARCHAR data types:

- **Casting to a CHAR data type.** You must use a size parameter. If you do not add a size parameter, a default of 30 is added. Syntax options appear in the following list:

- The recommended syntax is:

```
CAST(expr|NULL AS CHAR(n))
```

For example:

```
CAST(companyname AS CHAR(35))
```

- You can also use the following syntax:

```
CAST(expr|NULL AS data_type)
```

For example:

```
CAST(companyname AS CHAR)
```

Note:

If you use this syntax, the Oracle BI Server explicitly converts and stores as

```
CAST(expr |NULL AS CHAR(30))
```

- **Casting to a VARCHAR data type.** You must use a size parameter. If you omit the size parameter, you cannot save the change.

Examples

```
CAST(hiredate AS CHAR(40)) FROM employee
```

```
SELECT CAST(hiredate AS VARCHAR(40)), CAST(age AS double precision),  
CAST(hiredate AS timestamp), CAST(age AS integer) FROM employee
```

```
CAST("db"."table"."col" AS date)
```

CHOOSE

The CHOOSE function takes an arbitrary number of parameters and returns the first item in the list that the user has permission to view.

However, administrators must model the column permissions in the Oracle BI Administration Tool to enable this behavior. See [INDEXCOL](#) for an alternate method.

Syntax

```
CHOOSE(expr1, expr2, ..., exprN)
```

For example, a single query can be written to return security-based revenue numbers for the entire organization. The function could look like the following:

```
CHOOSE(L1-Revenue, L2-Revenue, L3-Revenue, L4-Revenue)
```

If the user issuing this function has access to the column L1-Revenue, then that column value would be returned. If the user does not have visibility to the column L1-Revenue but does have visibility to L2-Revenue, then L2-Revenue is returned.

IFNULL

The IFNULL function tests if an expression evaluates to a null value, and if it does, assigns the specified value to the expression.

Syntax

```
IFNULL(expr, value)
```

Where:

expr is the expression to evaluate.

value is the value to assign if the expression evaluates to a null value.

INDEXCOL

The INDEXCOL function can use external information to return the appropriate column for the logged-in user to see.

The Oracle BI Server handles this function in the following ways:

- **ODBC Procedures.** `NQSGetLevelDrillability` and `NQSGenerateDrillDownQuery` return the context-specific drill-down information based on the expression translated from INDEXCOL. This applies to both INDEXCOL expressions specified in the Logical SQL query and INDEXCOL expressions specified in a derived logical column.
- **Query Log and cache.** The Logical SQL query with INDEXCOL function appears in the SQL string in the query log. But the logical request does not show the INDEXCOL function because the Oracle BI Server translates INDEXCOL to one of the expressions in its expression list in the logical request generator.

The query cache uses the resulting translated expression for cache hit detection.

- **Usage Tracking.** Usage tracking inserts the Logical SQL query string with the INDEXCOL function.
- **Security.** As long as the user has the privileges to access the columns in the expression translated from INDEXCOL, then the query executes.

When the first argument to INDEXCOL is a session variable and if a default expression is expected to be returned even if the initialization block fails, then you should set a default value for the session variable. Otherwise, the query fails because the session variable has no value definition.

Syntax

`INDEXCOL(integer_literal, expr_list)`

Where:

`expr_list` equals the following:

`expr1 [, expr_list]`

The `INDEXCOL` function takes in an integer literal value as its first argument, followed by a variable length expression list and translates to a single expression from the expression list. The literal value is the 0-based index of the expression in the expression list to translate to. Consider the following expression:

`INDEXCOL(integer_literal, expr1, expr2, ...)`

If the literal value is 0, the above expression is equivalent to `expr1`. If the literal value is 1, then the value is equivalent to `expr2`, and so on.

The primary use case for `INDEXCOL` is for the first argument to contain a session variable. Specifying a constant literal would result in `INDEXCOL` always choosing the same expression.

Example With Hierarchy Levels

The fictional company, ABC, shows an example of hierarchy levels.

Company ABC has a geography dimension with the hierarchy Country, State, City. The CEO can access the Country level down to the City level, and the sales manager can access the State and City levels, and the sales people can only access the City level. The following table shows the back-end database for Company ABC.

User_Name	Title	Geo_level	Currency	Currency_col
Bob	CEO	0	US Dollars	0
Harriet	Sales Manager	1	Japanese Yen	1
Jackson	Sales Manager	1	Japanese Yen	1
Mike	Sales Person	2	Japanese Yen	1
Jim	Sales Person	2	US Dollars	0

The following steps illustrate one way to create a single query where each user sees the top level to which they have access:

- The administrator creates a new session variable called `GEOGRAPHY_LEVEL` that is populated by the following initialization block: `SELECT GEO_LEVEL from T where USER_NAME = ':USER'`.

This assumes that the Oracle BI Server instance has the same user names.

- Using `SELECT INDEXCOL(VALUEOF(NQ_SESSION.GEOGRAPHY_LEVEL), Country, State, City), Revenue FROM Sales`, the following occurs:

- Bob logs in and INDEXCOL translates to the Country column because the GEOGRAPHY_LEVEL session variable is 0. He gets the same result and can drill down on Country to State as if he had used `SELECT Country, Revenue FROM Sales`.
- Jackson logs in and INDEXCOL translates to the State column because the GEOGRAPHY_LEVEL session variable for Jackson is 1. He gets the same result and can drill down on State to City as if he had used `SELECT State, Revenue FROM Sales`.
- Mike logs in and INDEXCOL translates to the City column because the GEOGRAPHY_LEVEL session variable for Mike is 2. He gets the same result and cannot drill down on City as if he had used `SELECT City, Revenue FROM Sales`.

TO_DATETIME

The TO_DATETIME function converts string literals of dateTime format to a DateTime data type.

Syntax

```
TO_DATETIME('string1', 'DateTime_formatting_string')
```

Where:

string1 is the string literal you want to convert

DateTime_formatting_string is the DateTime format you want to use, such as yyyy.mm.dd hh:mi:ss. For this argument, yyyy represents year, mm represents month, dd represents day, hh represents hour, mi represents minutes, and ss represents seconds.

Examples

```
SELECT TO_DATETIME('2009-03-03 01:01:00', 'yyyy-mm-dd hh:mi:ss') FROM snowflakesales
```

```
SELECT TO_DATETIME('2009.03.03 01:01:00', 'yyyy.mm.dd hh:mi:ss') FROM snowflakesales
```

VALUEOF

Use the VALUEOF function to reference the value of a repository variable.

Repository variables are defined using the Oracle BI Administration Tool. You can use the VALUEOF function both in Expression Builder in the Administration Tool, and when you edit the SQL statements for an analysis from the Advanced tab of the Analysis editor in Administration Tool.

Syntax

Use the variables as arguments for the VALUEOF function. Refer to static repository variables by name. Variable names are case sensitive. For example, to use the value of a static repository variables named `prime_begin` and `prime_end`:

```
CASE WHEN "Hour" >= VALUEOF("prime_begin")AND "Hour" < VALUEOF("prime_end") THEN
'Prime Time' WHEN ... ELSE...END
```

You must refer to a dynamic repository variable by its fully qualified name. If you are using a dynamic repository variable, enclose the names of the initialization block and the repository variable in double quotes ("), separated by a period, and contained within parentheses. For example, to use the value of a dynamic repository variable named REGION contained in an initialization block named Region Security, use the following syntax:

```
SalesSubjectArea.Customer.Region = VALUEOF("Region Security"."REGION")
```

Begin the names of session variables with NQ_SESSION, separated by a period, and contained within parentheses, including the NQ_SESSION portion. For example, to use the value of a session variable named REGION, use the following syntax in Expression Builder or a filter:

```
"SalesSubjectArea"."Customer"."Region" = VALUEOF(NQ_SESSION.REGION)
```

If the variable name contains or begins with any characters other than ASCII alphanumeric characters ([A-Z] [a-z] [0-9]) or an underscore (_), then enclose the name in double quotes ("). Examples of such characters include a space, single quote, or double quote. The following example contains a single quote and a space:

```
"SalesSubjectArea"."Customer"."Region" = VALUEOF("NQ_SESSION"."Steven's Regions")
```

If the variable name that you specify contains double quotes ("), then escape the double quotes with a set of double quotes. For example:

```
"SalesSubjectArea"."Customer"."Region" = VALUEOF("NQ_SESSION"."""Top Sales"" Region")
```

Although using initialization block names with session variables might work, you should use NQ_SESSION. NQ_SESSION acts like a wildcard that matches all initialization block names. This lets you change the structure of the initialization blocks in a localized manner without impacting requests.

VARIABLE_REPLACE

The VARIABLE_REPLACE function queries a measure with the session variable value specified.

The VARIABLE_REPLACE function queries a measure with the session variable value specified. Use this function to query the measure using one or more session variable values. You can specify multiple name-value pairs. As long as the user has the permission to set the session variable, then the query executes.

You can only use this function when entering SQL at the command line using the nqcmd utility.

Syntax

```
VARIABLE_REPLACE(measure, variable_name, variable_value [, variable_name1, variable_value1, ...])
```

Where:

measure is a measure fact column that has a logical table source mapping containing session variables.

variable_name is a session variable name with the prefix NQ_SESSION.

variable_value is the substitute value for the session variable. String and integer are supported values. If the variable name contains a space, then it must be enclosed in double quotes.

Examples

This example shows the session variable being replaced with a string literal 'A'.

```
VARIABLE_REPLACE(Revenue, 'NQ_SESSION.VAR', 'A')
```

This example shows the session variable being replaced with the integer 1.

```
VARIABLE_REPLACE(Revenue, 'NQ_SESSION.VAR', 1)
```

Lookup Functions

A LOOKUP function is typically used in the Business Model and Mapping layer as an expression in a translated logical table column.

Multilingual schemas typically store translated fields in separate tables called lookup tables. Lookup tables contain translations for descriptor columns in several languages, while the base tables contain the data in the base language. *Lookup* is when a query joins the base table and lookup table to obtain the translated values for each row in the base table.

See the following:

- Supporting Multilingual Data in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition*
- About the LOOKUP Function Syntax in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition*

Database Functions

You can create requests by directly calling database functions or by using a logical column, in a logical table source in the metadata repository.

Users and administrators can create requests by directly calling database functions from either Oracle BI Answers, or by using a logical column, in the logical table source, within the metadata repository. Key uses for these functions include the ability to pass through expressions to get advanced calculations, as well as the ability to access custom written functions or procedures on the underlying database.

Note: You cannot use these functions with XML data sources.

- The EVALUATE_SUPPORT_LEVEL parameter in NQSCONFIG.INI controls the use of the EVALUATE family of database functions within Oracle BI Answers. Oracle recommends leaving EVALUATE_SUPPORT_LEVEL set to its default value of 0 to prevent the use of these functions within Oracle BI Answers. Setting EVALUATE_SUPPORT_LEVEL to a value of 1 or 2 enables users to insert

arbitrary SQL expressions into an analysis using Oracle BI Answers, which potentially compromises data access security. See Fusion Middleware System Administrator's Guide for Oracle Business Intelligence Enterprise Edition for more information about the EVALUATE_SUPPORT_LEVEL parameter.

- The EVALUATE_SUPPORT_LEVEL parameter in NQSCONFIG.INI does not control use of the EVALUATE family of database functions within the metadata repository. See NQSCONFIG.INI File Configuration Settings in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Functions include:

- [EVALUATE](#)
- [EVALUATE_ANALYTIC](#)
- [EVALUATE_AGGR](#)
- [EVALUATE_PREDICATE](#)

EVALUATE

The EVALUATE function, used for scalar functions that take input values and return an output value for a single row, passes the specified database function with optional referenced columns as parameters to the back-end data source for evaluation.

This function is intended for scalar calculations, and is useful when you want to use a specialized database function that is not supported by the Oracle BI Server, but that is understood by the underlying data source.

The embedded database function may require one or more columns. These columns are referenced by %1 ... %N within the function. You must list the actual columns after the function.

The ability to use EVALUATE is disabled by default. To enable support for this function, change the EVALUATE_SUPPORT_LEVEL parameter in NQSCONFIG.INI. See NQSCONFIG.INI File Configuration Settings in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Syntax

```
EVALUATE('db_function(%1...%N)' [AS data_type] [, column1, columnN])
```

Where:

db_function is any valid database function understood by the underlying data source.

data_type is an optional parameter that specifies the data type of the return result. Use this parameter whenever the return data type cannot be reliably predicted from the input arguments. However, do not use this parameter for type casting; if the function needs to return a particular data type, add an explicit cast. You can typically omit this parameter when the database-specific function has a return type not supported by the Oracle BI Server, but is used to generate an intermediate result that does not need to be returned to the Oracle BI Server.

column1 through *columnN* is an optional, comma-delimited list of column names.

Examples

This example shows an embedded database function.

```
SELECT EVALUATE('instr(%1, %2)', address, 'Foster City') FROM employees
```

Examples Using EVALUATE_AGGREGATE and EVALUATE to Leverage Unique Essbase Functions

The following examples use the EVALUATE_AGGREGATE and EVALUATE functions. Note that expressions are applied to columns in the logical table source that refers to the physical cube. Use EVALUATE_AGGREGATE to implement custom aggregations. For example, you may want to compare overall regional profit to profits for the top three products in the region. You can define a new measure to represent the profits for top three products resulting in the Logical SQL statement:

```
SELECT Region, Profit, EVALUATE_AGGREGATE('SUM(TopCount(%1.members, 3, %2), %3)',  
Products, Profit, Profit) Top_3_prod_Profit FROM SampleBasic
```

The Oracle BI Server generates the following expression for the custom aggregation:

```
member [Measures].[MS1] AS 'SUM(Topcount([Product].Generations(6).members,3,  
[Measures].[Profit]),[Measures].[Profit])'
```

Use the EVALUATE function on projected dimensions to implement scalar functions that are computed post-aggregation. EVALUATE may change the grain of the query, if its definition makes explicit references to dimensions (or attributes) that are not in the query.

For example, if you would like to see the Profits for the top five products ranked by Sales sold in a Region, after creating the applicable measure, the resulting Logical SQL statement is as follows

```
SELECT Region, EVALUATE('TopCount(%1.members, 5, %2)' as VARCHAR(20), Products,  
Sales), Profits FROM SampleBasic
```

The Oracle BI Server generates the following expression to retrieve the top five products:

```
set [Evaluate0] as '{Topcount([Product].Generations(6).members,5,[Measures].  
[Sales]) }'
```

EVALUATE_ANALYTIC

The EVALUATE_ANALYTIC function takes a row set, one or more rows, and returns results for each row in the set.

The EVALUATE_ANALYTIC function passes the specified database analytic function, with the optional referenced columns as parameters, to the back-end data source for evaluation. The functions are used to model SQL analytic functions, also known as, window functions.

The embedded database function might require one or more columns. These columns are referenced by %1 ... %N within the function. You must list the actual columns after the function.

The ability to use `EVALUATE_ANALYTIC` is disabled by default. To enable support for this function, change the `EVALUATE_SUPPORT_LEVEL` parameter in `NQSConfig.INI`. See `NQSConfig.INI File Configuration Settings` in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Syntax

```
EVALUATE_ANALYTIC('db_function(%1...%N)' [AS data_type] [, column1, columnN])
```

Where:

db_function is any valid database analytic function understood by the underlying data source.

data_type is an optional parameter that specifies the data type of the return result. Use this parameter when the return data type is not reliably predicted from the input arguments. However, do not use this parameter for type casting; if the function needs to return a particular data type, add an explicit cast. You can omit this parameter when the database-specific analytic function has a return type not supported by the Oracle BI Server, but is used to generate an intermediate result that does not need to be returned to the Oracle BI Server .

column1 through *columnN* is an optional, comma-delimited list of columns.

Examples

This example shows an embedded database analytic function.

```
EVALUATE_ANALYTIC('dense_rank() over(order by %1 )' AS INT,sales.revenue)
```

If the preceding example needs to return a double, then add an explicit cast, as follows:

```
CAST(EVALUATE_ANALYTIC('Rank(%1.dimension.currentmember, %2.members)',  
"Foodmart93"."Time"."Month" as Double)
```

EVALUATE_AGGR

The `EVALUATE_AGGR` function passes the specified database function, with optional referenced columns, as parameters to the back-end data source for evaluation.

This function is intended for aggregate functions with a `GROUP BY` clause.

The embedded database function may require one or more columns. These columns are referenced by `%1 ... %N` within the function. You must list the actual columns after the function.

The ability to use `EVALUATE_AGGR` is disabled by default. To enable support for this function, change the `EVALUATE_SUPPORT_LEVEL` parameter in `NQSConfig.INI`. See `NQSConfig.INI File Configuration Settings` in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Syntax

```
EVALUATE_AGGR('db_agg_function(%1...%N)' [AS data_type] [, column1, columnN])
```

Where:

db_agg_function is any valid aggregate database function understood by the underlying data source.

data_type is an optional parameter that specifies the data type of the return result. Use this parameter whenever the return data type is not reliably predicted from the input arguments. However, do not use this parameter for type casting; if the function needs to return a particular data type, add an explicit cast. You can omit this parameter when the database-specific function has a return type not supported by the Oracle BI Server, but is used to generate an intermediate result that does not need to be returned to the Oracle BI Server.

column1 through *columnN* is an optional, comma-delimited list of columns.

Example

```
EVALUATE_AGGR('REGR_SLOPE(%1, %2)', sales.quantity, market.marketkey)
```

EVALUATE_PREDICATE

The EVALUATE_PREDICATE function passes the specified database function with optional referenced columns as parameters to the back-end data source for evaluation.

The EVALUATE_PREDICATE is intended for functions that use a Boolean return type.

The embedded database function might require one or more columns. These columns are referenced by %1 ... %N within the function. The actual columns must be listed after the function.

Note:

EVALUATE_PREDICATE is not supported for use with Essbase data sources.

The ability to use EVALUATE_PREDICATE is disabled by default. To enable support for this function, change the EVALUATE_SUPPORT_LEVEL parameter in NQSConfig.INI. See NQSConfig.INI File Configuration Settings in *System Administrator's Guide for Oracle Business Intelligence Enterprise Edition* for more information.

Syntax

```
EVALUATE_PREDICATE('db_function(%1...%N)', [, column1, columnN])
```

Where:

db_function is any valid database function with a return type of Boolean that is understood by the underlying data source.

column1 through *columnN* is an optional, comma-delimited list of columns.

If you want to model a database function for comparison purposes, you should not use EVALUATE_PREDICATE. Use EVALUATE and put the comparison outside the function. For example, do not use EVALUATE_PREDICATE as follows:

```
EVALUATE_PREDICATE('dense_rank() over (order by 1% ) < 5', sales.revenue)
```

Use EVALUATE, as follows:

```
EVALUATE('dense_rank() over (order by 1% ) ', sales.revenue) < 5
```

Example

```
SELECT year, Sales AS DOUBLE,CAST(EVALUATE('OLAP_EXPRESSION(%1, 'LAG(units_cube_
sales, 1, time, time LEVELREL time_levelrel)')', OLAP_CALC) AS DOUBLE) FROM
"Global".Time, "Global"."Facts - sales" WHERE EVALUATE_PREDICATE('OLAP_
CONDITION(%1, 'LIMIT time KEEP ''1'', ''2'', ''3'', ''4'' ')
=1', OLAP_CALC) ORDER BY year;
```

Hierarchy Navigation Functions

The hierarchy navigation functions enable you to identify relationships between members of hierarchies.

The hierarchy navigation functions include:

- [DEPTH](#)
- [IDOF](#)
- [ISANCESTOR](#)
- [ISCHILD](#)
- [ISDESCENDANT](#)
- [ISLEAF](#)
- [ISPARENT](#)
- [ISROOT](#)
- [ISSIBLING](#)
- [PARENT](#)

The IDOF and ISLEAF functions apply to both level-based and parent-child hierarchies, while the other functions apply only to parent-child hierarchies.

See Working with Logical Dimensions in *Metadata Repository Builder's Guide for Oracle Business Intelligence Enterprise Edition* for information about level-based and parent-child hierarchies, including information about creating parent-child relationship tables (closure tables) for relational sources.

DEPTH

The DEPTH function returns an integer greater than 0 to indicate the depth of the member from the root member.

The DEPTH function returns an integer greater than 0 that indicates the depth of the member from the root member. The depth of a root member is 1.

Syntax

```
DEPTH(pc_presentation_hierarchy)
```

Where *pc_presentation_hierarchy* identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

The qualification term ("subject_area" .) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

Example

```
DEPTH("employees"."emp_hierarchy")
```

IDOF

The IDOF function returns a representation of the member key column values for a given member in a parent-child hierarchy, or a representation of the level key column values for a given level in a level-based hierarchy.

If the given hierarchy or level has a key that contains exactly one logical column, the function returns the value of that logical column. If the given level has a key that contains more than one logical column, the function returns a concatenated string. For example:

```
"keycolumn1value"."keycolumn2value"."keycolumn3value"...
```

Note:

The parent-child hierarchies do not have keys that contain more than one logical column.

If the given hierarchy or level has no key, an error is displayed.

Syntax

For parent-child hierarchies:

```
IDOF(pc_presentation_hierarchy)
```

Where *pc_presentation_hierarchy* identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

For level-based hierarchies:

```
IDOF(level)
```

Where *level* is a fully-qualified presentation level that is based on a level in a level-based dimension, as follows:

```
"subject_area"."presentation_table"."presentation_hierarchy"."presentation_level"
```

Examples

```
IDOF("hr"."employees"."emp_hierarchy")
```

```
IDOF("market_data"."products"."product"."product")
```

ISANCESTOR

The ISANCESTOR function enables finding the ancestors of one or more members of a parent-child hierarchy, that is all the ancestors of the members, or the ancestors at a specified hierarchical distance from the members.

Each member of the parent-child hierarchy is compared with the specified members to determine if it is an ancestor. The ISANCESTOR function returns the Boolean value True for each ancestor of the specified members, else it returns False.

When you use the function to find ancestors of multiple members, then an OR operation is performed. That is, find the ancestors of Joe or Juana. If you need an AND operation, then call the ISANCESTOR function multiple times, once for each member.

You can use the ISANCESTOR function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISANCESTOR function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column. Note that the syntax of the function depends on where you are using it.

Presentation Layer Syntax

```
ISANCESTOR(pc_presentation_hierarchy, member_identifiers [, distance])
```

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

The qualification term, "subject_area" ., is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

member_identifiers is the string or numeric literals that identify the one or more members in *pc_presentation_hierarchy* . Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

distance (*optional*) is a positive integer that identifies the distance from the specified members to the parent-child hierarchy level at which to search for ancestors.

By default, if *distance* is not specified, the ISANCESTOR function searches the current parent-child level that contains *member_identifiers* and all levels above.

Business Model and Mapping Layer Syntax

```
ISANCESTOR(logical_dimension, member_identifiers [, distance])
```

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

```
"business_model"."dimension_name"
```

The qualification term, "business_model" ., is optional unless there are multiple dimensions with the same name in different business models.

member_identifiers is the string or numeric literals that identify the one or more members in *logical_dimension* . Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

distance (optional) is a positive integer that identifies the distance from the specified members to the parent-child hierarchy level at which to search for ancestors.

By default, if *distance* is not specified, the ISANCESTOR function searches the current parent-child level that contains *member_identifiers* and all levels above.

Example

The following example selects all the ancestor employees of the employee Joe in a parent-child hierarchy. The returned list includes the employee Joe.

```
SELECT emp_name
FROM "employees"
WHERE ISANCESTOR("employees"."emp_hierarchy", 'Joe')
```

ISCHILD

The ISCHILD function enables finding the children of one or more members of a parent-child hierarchy, that is, all the members that are one hierarchical level below the specified members.

The ISCHILD function enables finding the children of one or more members of a parent-child hierarchy, that is, all the members that are one hierarchical level below the specified members.

Note:

The ISCHILD function is the same as the ISDESCENDANT function with a distance parameter of 1.

The ISCHILD function returns the Boolean value True for each child of the specified members, else it returns False.

When you use the function to find children of multiple members, then an OR operation is performed. That is, find the children of Joe or Juana. If you need an AND operation, then call the ISCHILD function multiple times, once for each member.

You can use the ISCHILD function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISCHILD function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column.

Note:

The syntax of the function depends on where you are using it.

Presentation Layer Syntax

`ISCHILD(pc_presentation_hierarchy, member_identifiers)`

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

"subject_area"."presentation_table"."pc_presentation_hierarchy"

The qualification term ("subject_area" .) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

member_identifiers is the string or numeric literals that identify the one or more members in *pc_presentation_hierarchy*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Business Model and Mapping Layer Syntax

`ISCHILD(logical_dimension, member_identifiers)`

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

"business_model"."dimension_name"

The qualification term ("business_model" .) is optional unless there are multiple dimensions with the same name in different business models.

member_identifiers is the string or numeric literals that identify the one or more members in *logical_dimension*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Example

The following example selects all the children of the employee Joe in a parent-child hierarchy.


```
SELECT emp_name
FROM "employees"
WHERE ISCHILD("employees"."emp_hierarchy", 'Joe')
```

ISDESCENDANT

The ISDESCENDANT function enables finding the descendants of one or more members of a parent-child hierarchy, all the descendants of the members, or the descendants at a specified hierarchical distance from the members.

Each member of the parent-child hierarchy is compared with the specified members to determine if it is a descendant. The ISDESCENDANT function returns the Boolean value True for each descendant of the specified members, else it returns False.

When you use the function to find descendants of multiple members, then an OR operation is performed. That is, find the descendants of Joe or Juana. If you need an AND operation, then call the ISDESCENDANT function multiple times, once for each member.

You can use the ISDESCENDANT function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISDESCENDANT function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column. Note that the syntax of the function depends on where you are using it.

Presentation Layer Syntax

```
ISDESCENDANT(pc_presentation_hierarchy, member_identifiers [, distance])
```

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

The qualification term ("*subject_area*".) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

member_identifiers is the string or numeric literals that identify the one or more members in *pc_presentation_hierarchy*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

distance (*optional*) is a positive integer, that identifies the distance from the specified members to the parent-child hierarchy level at which to search for descendants.

By default, if *distance* is not specified, the ISDESCENDANT function searches the current parent-child level that contains *member_identifiers* and all levels below.

Business Model and Mapping Layer Syntax

```
ISDESCENDANT(logical_dimension, member_identifiers [, distance])
```

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

```
"business_model"."dimension_name"
```

The qualification term ("business_model" .) is optional unless there are multiple dimensions with the same name in different business models.

member_identifiers is the string or numeric literals that identify the one or more members in *logical_dimension*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

distance (*optional*) is a positive integer that identifies the distance from the specified members to the parent-child hierarchy level at which to search for descendants.

By default, if *distance* is not specified, the ISDESCENDANT function searches the current parent-child level that contains *member_identifiers* and all levels below.

Example

The following example selects all the descendant employees of the employee Joe in a parent-child hierarchy. The returned list includes the employee Joe.

```
SELECT emp_name
FROM "employees"
WHERE ISDESCENDANT("employees"."emp_hierarchy", 'Joe')
```

ISLEAF

The ISLEAF function applies to both level-based and parent-child hierarchies.

For both types of hierarchy, a leaf member is defined as a member that has no child members.

Each member of the hierarchy is examined to determine if it is a leaf member. The ISLEAF function returns the Boolean value True for each leaf member, else it returns False.

You can use the ISLEAF function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISLEAF function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column.

Note:

The syntax of the function depends on where you are using it.

Presentation Layer Syntax

```
ISLEAF(presentation_hierarchy)
```

Where:

presentation_hierarchy identifies the fully qualified presentation hierarchy, either level-based or parent-child, as follows:

```
"subject_area"."presentation_table"."presentation_hierarchy"
```

The qualification term ("subject_area" .) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

Business Model and Mapping Layer Syntax

```
ISLEAF(logical_dimension)
```

Where:

logical_dimension identifies the fully qualified dimension that contains the hierarchy you want to navigate, either level-based or parent-child, as follows:

```
"business_model"."dimension_name"
```

The qualification term ("business_model" .) is optional unless there are multiple dimensions with the same name in different business models.

Example

The following example selects all the employees in a hierarchy that are leaf members, that is, the employees who have no members below them in the hierarchy.

```
SELECT emp_name
FROM "employees"
WHERE ISLEAF("employees"."emp_hierarchy")
```

ISPARENT

The ISPARENT function enables finding the parents of one or more members of a parent-child hierarchy, that is, all the members that are one hierarchical level above the specified members.

Note:

The ISPARENT function is the same as the ISANCESTOR function with a distance parameter of 1.

The ISPARENT function returns the Boolean value True for each parent of the specified members, else it returns False.

When you use the function to find parents of multiple members, then an OR operation is performed. That is, find the parents of Joe or Juana. If you need an AND operation, then call the ISPARENT function multiple times, once for each member.

You can use the ISPARENT function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISPARENT function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column.

Note:

The syntax of the function depends on where you are using it.

Presentation Layer Syntax

`ISPARENT(pc_presentation_hierarchy, member_identifiers)`

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

"subject_area"."presentation_table"."pc_presentation_hierarchy"

The qualification term ("subject_area" .) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

member_identifiers is the string or numeric literals that identify the one or more members in *pc_presentation_hierarchy*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Business Model and Mapping Layer Syntax

`ISPARENT(logical_dimension, member_identifiers)`

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

"business_model"."dimension_name"

The qualification term ("business_model" .) is optional unless there are multiple dimensions with the same name in different business models.

member_identifiers is the string or numeric literals that identify the one or more members in *logical_dimension*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Example

The following example selects all the parents of the employee Joe in a parent-child hierarchy.

```
SELECT emp_name
FROM "employees"
WHERE ISPARENT("employees"."emp_hierarchy", 'Joe')
```

ISROOT

A presentation hierarchy member is a root member if it does not have any ancestors above it in a parent-child presentation hierarchy.

Each member of the parent-child hierarchy is examined to determine if it is a root member. The ISROOT function returns the Boolean value True for each root member, else it returns False.

You can use the ISROOT function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISROOT function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column.

Note:

The syntax of the function depends on where you are using it.

Presentation Layer Syntax

ISROOT(*pc_presentation_hierarchy*)

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

"subject_area"."presentation_table"."pc_presentation_hierarchy"

The qualification term ("subject_area".) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

Business Model and Mapping Layer Syntax

ISROOT(*logical_dimension*)

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

"business_model"."dimension_name"

The qualification term ("business_model".) is optional unless there are multiple dimensions with the same name in different business models.

Example

The following example selects all the employees in a parent-child hierarchy that are root members, that is, the employees who have no ancestors above them in the hierarchy.

```
SELECT emp_name
FROM "employees"
WHERE ISROOT("employees"."emp_hierarchy")
```

ISSIBLING

The ISSIBLING function enables finding the siblings of one or more members of a parent-child hierarchy, that is, all the members that have the same parent as the specified members.

The sibling members can be parents themselves of other members. Root members with null parents are siblings of each other.

The ISSIBLING function returns the Boolean value True for each sibling of the specified members, else it returns False.

When you use the function to find siblings of multiple members, then an OR operation is performed. That is, find the siblings of Joe or Juana. If you need an AND operation, then call the ISSIBLING function multiple times, once for each member.

You can use the ISSIBLING function in a query both within CASE statements and in WHERE clause conditions.

You can use the ISSIBLING function in both Presentation layer queries, and in the Business Model and Mapping layer, for example, when creating a derived column.

Note:

The syntax of the function depends on where you are using it.

Presentation Layer Syntax

```
ISSIBLING(pc_presentation_hierarchy, member_identifiers)
```

Where:

pc_presentation_hierarchy identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

The qualification term ("*subject_area*".) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

member_identifiers is the string or numeric literals that identify the one or more members in *pc_presentation_hierarchy*. Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Business Model and Mapping Layer Syntax

```
ISSIBLING(logical_dimension, member_identifiers)
```

Where:

logical_dimension identifies the fully qualified dimension that contains the parent-child hierarchy, as follows:

```
"business_model"."dimension_name"
```

The qualification term ("business_model" .) is optional unless there are multiple dimensions with the same name in different business models.

member_identifiers is the string or numeric literals that identify the one or more members in *logical_dimension* . Separate multiple literals with commas and enclose the group in parentheses, such as (2, 3). The type of literal depends on the data type of the dimension level keys.

Example

The following example selects all the siblings of the employee Joe in a parent-child hierarchy.

```
SELECT emp_name
FROM "employees"
WHERE ISSIBLING("employees"."emp_hierarchy", 'Joe')
```

PARENT

The PARENT function returns the member key column value for the parent of a given member in a parent-child hierarchy.

The PARENT function returns the member key column value for the parent of a given member in a parent-child hierarchy.

If the given hierarchy has no key, an error is displayed.

Syntax

```
PARENT(pc_presentation_hierarchy)
```

Where *pc_presentation_hierarchy* identifies the fully qualified parent-child presentation hierarchy, as follows:

```
"subject_area"."presentation_table"."pc_presentation_hierarchy"
```

The qualification term ("subject_area" .) is optional unless there are multiple presentation tables or presentation hierarchies with the same name in different subject areas.

Example

```
PARENT("employees"."emp_hierarchy")
```

System Functions

These system functions return values related to the session.

Functions include:

- [USER](#)
- [DATABASE](#)

USER

The USER function returns the user name for the Oracle BI repository to which you are logged on.

Syntax

```
USER( )
```

DATABASE

The DATABASE function returns the name of the default subject area.

Syntax

```
DATABASE( )
```

Advanced Analytics Internal Logical SQL Functions

Oracle BI Server supports Logical SQL functions allowing Visual Analyzer users to explore data using models such as binning and trend lines.

These Logical SQL functions are also available to Presentation Services users. This section contains the following topics:

- [Comparison of BIN and WIDTH_BUCKET Functions](#)
- [BIN Function](#)
- [WIDTH_BUCKET Function](#)
- [TRENDLINE Function](#)

Comparison of BIN and WIDTH_BUCKET Functions

Oracle BI Server supports the BIN and WIDTH_BUCKET Logical SQL functions. These functions share some of the same parameters and syntax, however, the usage of each functions is different.

The following are the differences between the BIN and WIDTH_BUCKET functions:

- WIDTH_BUCKET function
 - The WIDTH_BUCKET function is not treated as a new dimensional attribute for the purposes of aggregation. Instead, the WIDTH_BUCKET function is applied on top of the query result similar to the other display functions such as RANK, TOPN, BOTTOMN, NTILE, PERCENTILE, MAVG, and MEDIAN.
 - Use the WIDTH_BUCKET function when you want to compute a discrete set of buckets on top of an already aggregated query result set.

- The BY clause is always optional in the WIDTH_BUCKET function. If the BY clause is omitted from the WIDTH_BUCKET function, then the function operates over the entire result set.
- The BY clause of the WIDTH_BUCKET function defines the groups in the query result set over which the WIDTH_BUCKET calculation is applied. The buckets within different groups are calculated independently.
- BIN function
 - The BIN function is treated as a new dimensional attribute for the purposes of aggregation.
 - The BY clause of the BIN function defines the grain at which the binned expression is evaluated prior to binning. If the binned expression is a measure, then the measure is grouped at the grain specified in the BY clause before being binned.
 - The BY clause of the BIN function is mandatory if the binned expression is a measure. Otherwise, for non-measure expressions, the BY clause is optional.
 - Use the BIN function when you want to compute a set of discrete buckets on top of a continuous valued attribute or measure and you want to treat that new set of discrete buckets as if it were a new dimension attribute that is intended to be included in the GROUP BY clause of other base measures in the query.

BIN Function

With the BIN function, you can select any numeric attribute (INT, FLOAT, DOUBLE, NUMERIC) from a dimension or fact table or a measure containing the data values and place them into a discrete number of bins.

The BIN function classifies a given numeric expression into a specified number of equal width buckets. The function can return either the bin number or one of the two end points of the bin interval. The BIN function output is used as a GROUP BY expression for other measures included in the query.

For example, suppose the user is analyzing the sales performance for a retail store. One way to look at sales is in the form of a histogram so that you can see sales less than \$500, between \$500 and \$1000, and so on. To build a histogram first you would classify the sales total values into bins.

The BIN function is treated like a new dimension attribute for purposes such as aggregation, filtering, and drilling. All of these operations are supported on BIN expressions.

The WIDTH_BUCKET function is an alternative to the BIN function for users who require bucketing on top of the final query result set as a display function.

Syntax

```
BIN(numeric_expr [BY grain_expr1, ..., grain_exprN] [WHERE condition]
  INTO number_of_bins BINS [BETWEEN min_value AND max_value]
  [RETURNING { NUMBER | RANGE_LOW | RANGE_HIGH }])
```

Where:

numeric_expr indicates the measure or numeric attribute to bin.

BY *grain_expr1*, ..., *grain_exprN* indicates a list of expressions that define the grain used in the *numeric_expr* that is calculated before the numeric values are assigned to bins. This clause is required for measure expressions and is optional for attribute expressions.

WHERE condition indicates a filter condition to apply to the *numeric_expr* before the numeric values are assigned to bins.

INTO *number_of_bins* indicates the number of bins to return. The default is 10.

BETWEEN *min_value* AND *max_value* indicates the minimum and maximum values used for the end points of the outermost bins.

RETURNING indicates a filter condition to apply to the *numeric_expr* before the numeric values are assigned to bins. You can use the following options:

- RETURNING NUMBER indicates the return value is the bin number, for example, 1, 2, 3, 4. This is the default condition.
- RETURNING RANGE_LOW indicates the lower value of the bin interval.
- RETURNING RANGE_HIGH indicates the higher value of the bin interval.

Multiple Years Example

In this example, the filter Year IN (2013, 2014, 2015) applies to the calculation of revenue, but it does not affect the UnitPrice subquery used to calculate the bins by product.

```
SELECT Year, BIN(UnitPrice BY ProductName), Revenue
FROM SampleSalesSubjectArea
WHERE Year IN (2013, 2014, 2015);
```

Year In Filter Example

In this example, the filter Year = 2014 is explicitly applied to the bin calculation and affects the QtySold expression that is used to define the bins. The Year IN (2013, 2014, 2015) filter is applied to the calculation of Revenue but does not affect the bin definition.

This query returns the low and high end points of the bin intervals.

```
SELECT Year,
BIN(QtySold BY ProductName WHERE Year = 2014 INTO 20 BINS RETURNING RANGE_LOW),
BIN(QtySold BY ProductName WHERE Year = 2014 INTO 20 BINS RETURNING RANGE_HIGH),
Revenue
FROM SampleSalesSubjectArea
WHERE Year IN (2013, 2014, 2015);
```

QtySold Example

In this example, the measure, QtySold, is computed at the City grain. Each city, regardless of its region, is assigned to bins 1, 2, 3, 4, or 5.

The bin query containing the bin number (1 through 5) and the City primary key are joined with the main query on the City key. The main query is grouped by Region and the City bin (1 through 5).

Some regions might not have each city bin number represented in the final result set. For example, the rows for the region East might contain bins 1, 3, 4, while the region West might only have bins 1, 2, 4, 5.

```
SELECT Region, BIN(QtySold BY City INTO 5 BINS), Revenue
FROM SampleSalesSubjectArea;
```

Age Attribute Example

In this example, because *Age* is an attribute, the *BY* clause is omitted. In this case, the grain used to compute the bins is determined by the attribute's associated level defined in the metadata.

```
SELECT BIN(Age), Revenue
FROM SampleSalesSubjectArea;
```

Age and Customer Example

In this example, the *BY* clause explicitly states the grain where the *Age* attribute is computed. If the attribute contains multiple values at the grain, then the Oracle BI Server grain check detects the error condition and displays an error message indicating which tuple contains the multiple values.

```
SELECT BIN(Age BY Customer), Revenue
FROM SampleSalesSubjectArea;
```

WIDTH_BUCKET Function

The `WIDTH_BUCKET` function assigns a given numeric expression into a specified number of equal width buckets.

The `WIDTH_BUCKET` function operates on top of the base query result set as a display function. The function can return either the bin number or one of the two end points of the bin interval.

Syntax

The syntax of the `WIDTH_BUCKET` function uses simple comma-separated arguments. The only nested clause supported by the `WIDTH_BUCKET` function is the *BY* clause.

```
WIDTH_BUCKET(numeric_expr, { NUMBER | RANGE_LOW | RANGE_HIGH },
number_of_bins, [min_value, max_value] [BY expr1, ..., exprN])
```

Where:

numeric_expr indicates the measure or numeric attribute to bin.

`NUMBER` indicates that the return value is the bin number, for example, 1, 2, 3, 4. `Number` is the default return value.

`RANGE_LOW` indicates the lower value of the bin interval.

`RANGE_HIGH` indicates the higher value of the bin interval.

number_of_bins indicates the number of bins to return. The default is 10.

The *min_value* and *max_value* indicate the minimum and maximum values used for the end points of the outermost bins. If the *min_value* and *max_value* conditions are omitted, then the function determines the end points automatically.

BY *expr1*, ..., *exprN* indicates an optional list of expressions that define the groups in the query result set from which the WIDTH_BUCKET calculation is applied. The bucket intervals within different groups are calculated independently.

Year IN Filter Example

In this example, the filter Year IN (2013, 2014, 2015) applies to the entire result set. The revenue measure is computed with the Year, Category grain. The WIDTH_BUCKET is calculated using the filtered and aggregated revenue measure in the Year, Category grain.

```
SELECT Year, Category, Revenue, WIDTH_BUCKET(Revenue, RANGE_HIGH, 5)
FROM SampleSalesSubjectArea
WHERE Year IN (2013, 2014, 2015);
```

Revenue Measure Example

In this example, the revenue measure is computed at the Year, Category grain, and the WIDTH_BUCKET is calculated on top of the aggregated revenue measure.

The WIDTH_BUCKET function specifies 10 buckets between the values 0 and 1,000,000. The result is 10 buckets each with an interval of 100,000. Any revenue values outside of the range (0, 1,000,000) are placed in to overflow buckets. If there are revenue values below 0 or above 1,000,000, the result could include buckets 11 or 12 for the low overflow values and the high overflow values.

```
SELECT Year, Category, Revenue, WIDTH_BUCKET(Revenue, NUMBER, 10, 0, 1000000)
FROM SampleSalesSubjectArea;
```

Independent Bucket Intervals Example

In this example, the WIDTH_BUCKET is partitioned by Year so that the bucket intervals are computed independently for each Year. The query might not make much sense unless the results are divided into multiple visualizations by Year. For example, when using a trellis chart, the bucket intervals from different years are not necessarily aligned to the same end points.

```
SELECT Year, Category, Revenue, QtySold
WIDTH_BUCKET(Revenue, RANGE_LOW, 5 BY Year) as Low,
WIDTH_BUCKET(Revenue, RANGE_HIGH, 5 BY Year) as High
FROM SampleSalesSubjectArea;
```

TRENDLINE Function

The TRENDLINE function measures data across time, and shows a metric by ordered records line chart.

The TRENDLINE function can model data as linear and exponential regression.

Syntax

```
TRENDLINE( <numeric_expr>, ( [<series>] ) BY ( [<partitionBy>] ), <model_type>,
<result_type>, [number_of_degrees] )
```

Where:

numeric_expr indicates the data to trend. The *numeric_expr* is usually a measure column and the Y-axis.

series indicates the X-axis. This is a list of <valueExp> <orderByDirection>, where <valueExp> is a dimension column and <orderByDirection> is *ASC*, ascending, or *DES*, descending. The default is *ASC*. You cannot use an arbitrary combination of numeric columns.

partitionBy indicates the control break for the trendline.

model_type indicates the type of model to use. You can only specify *LINEAR*.

result_type indicates the type of output. You can specify *VALUE* or *MODEL*. *VALUE* returns the regression Y values given X in the fit. *MODEL* returns the parameters in a JSON format string.

number_of_degrees is used in polynomial models, only. The *number_of_degrees* parameter is optional.

Using Sales.Revenue and day.cal_year Example

This example shows a trend line using *Sales.Revenue* as the Y-axis and *day.cal_year* as the X-axis. The regression method is linear regression and the returning type is value. The *day.cal_year* is a primary level key from a time dimension.

```
SELECT day.cal_year, Sales.Revenue, TRENDLINE(Sales.Revenue, (day.cal_year),
(), 'LINEAR', 'VALUE')
FROM TimeSeriesTesting;
```

Product.CategoryID Example

This example shows a separate trend line using *Sales.Revenue* as the Y-axis and *day.cal_year* as the X-axis for each *Product.CategoryID*.

```
SELECT Product.CategoryID, day.cal_year, Sales.Revenue, TRENDLINE(Sales.Revenue,
(day.cal_year), (Product.CategoryID), 'LINEAR', 'VALUE')
FROM TimeSeriesTesting; //single trendline with control_break
```

Multiple Trendline Function Example

This example shows a multiple trendline function in the same query. This example uses *Sales.Revenue* as Y-axis and (*day.cal_year*, *day.cal_qtr*) as X-axis.

```
SELECT day.cal_year, day.cal_qtr, Sales.Revenue, TRENDLINE(Sales.Revenue,
(day.cal_year, day.cal_qtr), (), 'LINEAR', 'VALUE'), TRENDLINE(Sales.Revenue,
(day.cal_year, day.cal_qtr), (), 'LINEAR', 'MODEL')
FROM TimeSeriesTesting
order by 1; //multiple trendline functions
```

Separate Trendlines Example

This example shows a separate trendline using *Sales.Revenue* as Y-axis and *day.cal_year* as X-axis for each *Product.CategoryID*. Although the *Shipper.ShipperID* is in the query projection list, the trendline does not use the *Shipper.ShipperID* for computation.

```
SELECT Product.CategoryID, Shipper.ShipperID, day.cal_year, Sales.Revenue,
TRENDLINE(aggregate(Sales.Revenue by day.cal_year, Product.CategoryID),
(day.cal_year), (Product.CategoryID), 'LINEAR', 'Value')
FROM TimeSeriesTesting
order by 1, 3; //Query grain can be lower than Trendline Y-axis grain.
```

Single Numeric Column Example

This example uses a single numeric column as Trendline X-axis.

```
SELECT employee.country, category.categoryid, sales.qtytsold,  
TRENDLINE(sales.qtytsold, (category.categoryid), (employee.country), 'LINEAR', 'VALUE')  
FROM TimeSeriesTesting  
order by 1, 2;
```

Use a single numeric column as the Trendline's X-axis.

Advanced Analytics External Logical SQL Functions

Oracle BI Server supports external Logical SQL functions that allow Visual Analyzer users to explore data with models such as forecast, cluster, and outlier.

The external Logical SQL functions described in this section are also available to Presentation Services users.

External Logical SQL functions feed the input data to the standalone R-process or ORE. Depending upon your environment and query needs, your environment must include either the R or ORE external engine before you can successfully use these functions. See *Installing R and Oracle R Enterprise for External Logical SQL in the System Administrator's Guide for Oracle Business Intelligence Enterprise Edition*.

This section contains the following topics:

- [EVALUATE_SCRIPT Function](#)
- [FORECAST Function](#)
- [CLUSTER Function](#)
- [OUTLIER Function](#)
- [REGR Function](#)

EVALUATE_SCRIPT Function

The EVALUATE_SCRIPT function executes an R script as specified in the `script_file_path`, passing in one or more columns or literal expressions as input. The output of the function is determined by the `column_name`.

Syntax

```
EVALUATE_SCRIPT(script_file_path,column_name,options, [runtime_binded_column_options])
```

Where:

script_file_path indicates the script XML file path. For example, `filerepo//obiee.TimeSeriesForecast.xml`.

column_name indicates the column name upon which to forecast.

options is a string list of names or value pairs separated by a semi-colon (;). For example:

```
'algorithm=GLM;CustomerID=%1;ActualRevenue=%2;YearsAsCustomer=%3'
```

runtime_binded_column_options indicates an optional variable list of column expression. You can specify one or more columns.

Example 1

This example shows how to use the EVALUATE_SCRIPT command.

```
EVALUATE_SCRIPT(
'filerepo://PAFScore_bounded.xml',
'predictionValue',
'algorithm=GLM;CustomerID=%1;ActualRevenue=%2;YearsAsCustomer=%3',
'A - Sample Sales"."Customers"."C0 Customer Number",
'A - Sample Sales"."Datamining"."M01- Actual Revenue",
'A - Sample Sales"."Customers"."C20 Years as Customer"
) PredictedRevenue
```

Example 2

This example shows how to use the EVALUATE_SCRIPT function to call the Clustering.xml script.

```
SELECT
'A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
'A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
EVALUATE_SCRIPT(
'filerepo://obiee.Clustering.xml',
'clusterId',
'algorithm=kmeans;numClusters=3;id=%1;arg1=%2;',
'A - Sample Sales"."Customers"."C0 Customer Number" ,
'A - Sample Sales"."Datamining"."M02- Expected Rev") ClusterId,
EVALUATE_SCRIPT(
'filerepo://obiee.Clustering.xml',
'clusterName',
'algorithm=kmeans;numClusters=3;id=%1;arg1=%2;',
'A - Sample Sales"."Customers"."C0 Customer Number" ,
'A - Sample Sales"."Datamining"."M02- Expected Rev") ClusterName,
EVALUATE_SCRIPT(
'filerepo://obiee.Clustering.xml',
'clusterSize',
'algorithm=kmeans;numClusters=3;id=%1;arg1=%2;',
'A - Sample Sales"."Customers"."C0 Customer Number" ,
'A - Sample Sales"."Datamining"."M02- Expected Rev") ClusterSize,
EVALUATE_SCRIPT(
'filerepo://obiee.Clustering.xml',
'distanceFromCenter',
'algorithm=kmeans;numClusters=3;id=%1;arg1=%2;',
'A - Sample Sales"."Customers"."C0 Customer Number" ,
'A - Sample Sales"."Datamining"."M02- Expected Rev") DistanceFromCenter
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers".
"C0 Customer Number" < 1000;
```

Example 3

This example shows how to use the EVALUATE_SCRIPT function to call the TimeSeriesForecast.xml script.

```
SELECT
'A - Sample Sales"."Time"."T00 Calendar Date" Day1,
'A - Sample Sales"."Base Facts"."1- Revenue" Target,
EVALUATE_SCRIPT(
'filerepo://obiee.TimeSeriesForecast.xml',
'forecast',
```

```
'numPeriods=7;predictionInterval=90;timeDay=%1;target=%2;',
"A - Sample Sales"."Time"."T00 Calendar Date",
"A - Sample Sales"."Base Facts"."1- Revenue") ForecastedRevenue,
EVALUATE_SCRIPT(
'filerepo://obiee.TimeSeriesForecast.xml',
'high',
'numPeriods=7;predictionInterval=90;timeDay=%1;target=%2;',
"A - Sample Sales"."Time"."T00 Calendar Date",
"A - Sample Sales"."Base Facts"."1- Revenue") High,
EVALUATE_SCRIPT(
'filerepo://obiee.TimeSeriesForecast.xml',
'low',
'numPeriods=7;predictionInterval=90;timeDay=%1;target=%2;',
"A - Sample Sales"."Time"."T00 Calendar Date",
"A - Sample Sales"."Base Facts"."1- Revenue") Low
FROM "A - Sample Sales" order by 1;
```

Example 4

This example shows how to use EVALUATE_SCRIPT function to call the Outliers.xml.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
EVALUATE_SCRIPT(
'filerepo://obiee.Outliers.xml',
'isOutlier',
'algorithm=mvoutlier;id=%1;arg1=%2;arg2=%3;',
"A - Sample Sales"."Customers"."C0 Customer Number" ,
"A - Sample Sales"."Datamining"."M02- Expected Rev",
"A - Sample Sales"."Customers"."C21 Customer Age") IsOutlier,
EVALUATE_SCRIPT(
'filerepo://obiee.Outliers.xml',
'distance',
'algorithm=mvoutlier;id=%1;arg1=%2;arg2=%3;',
"A - Sample Sales"."Customers"."C0 Customer Number" ,
"A - Sample Sales"."Datamining"."M02- Expected Rev",
"A - Sample Sales"."Customers"."C21 Customer Age") Distance
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers".
"C0 Customer Number" < 1000
```

FORECAST Function

The FORECAST function creates a time-series model of the specified measure over the series using Exponential Smoothing or ARMA.

This function outputs a prediction for the set of periods as specified by numPeriods.

Syntax

```
FORECAST(numeric_expr, ([series]), output_column_name, options,
[runtime_binded_options]))
```

Where:

numeric_expr indicates the measure to forecast, for example, revenue data to forecast.

series indicates the time grain at which the forecast model is built. This is a list of one or more time dimension columns. If you omit *series*, then the time grain is determined from the query.

output_column_name indicates the output column. The valid values are *forecast*, *low*, *high*, and *predictionInterval*.

options indicates a string list of name/value pairs separated by a semi-colon (;). The value can include %1 ... %N specified in *runtime_binded_options*.

runtime_binded_options indicates a comma separated list of runtime-binded columns and options.

FORECAST Function Options

The following table list available options to use with the FORECAST function.

Option Name	Description	Values
numPeriods	The number of periods to forecast	Integer
predictionInterval	The confidence level for the prediction.	0 to 100, where higher values specify higher confidence
modelType	The model to use for forecasting.	ARIMA ETS
useBoxCox	If TRUE, then use Box-Cox transformation.	TRUE FALSE
lambdaValue	The Box-Cox transformation parameter. Ignore if NULL or when useBoxCox is FALSE. Otherwise the data is transformed before the model is estimated.	Not applicable
trendDamp	This is a parameter for ETS model. If TRUE, then use damped trend. If NULL, then try both damped and non-damped trend and choose the one that is optimal.	Not applicable
errorType	This is a parameter for ETS model.	additive("A") multiplicative("M") automatically selected ("Z")

Option Name	Description	Values
trendType	This is a parameter for ETS model.	none("N") additive("A") multiplicative("M") automatically selected ("Z")
seasonType	This is a parameter for ETS model.	none("N") additive("A") multiplicative("M") automatically selected("Z")
modelParamIC	The information criterion (IC) used in the model selection.	ic_auto ic_aicc ic_bic ic_auto (this is the default)

Example 1

This example selects revenue forecast by day.

```
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue" Target,
("A - Sample Sales"."Time"."T00 Calendar Date"),'forecast',
'numPeriods=30;predictionInterval=70;') ForecastedRevenue
```

Example 2

This example selects revenue forecast by year and quarter.

```
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue",
("A - Sample Sales"."Time"."T01 Year" timeYear, "A - Sample Sales"."Time"."T02
Quarter" TimeQuarter),'forecast', 'numPeriods=30;predictionInterval=70;')
ForecastedRevenue
```

Example 3

This example selects revenue forecast by year and quarter using a custom forecast script XML file.

```
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue",
("A - Sample Sales"."Time"."T01 Year" timeYear, "A - Sample Sales"."Time"."T02
Quarter" TimeQuarter),'forecast',
'numPeriods=30;predictionInterval=70;Script=filerepo://PAFTimeSeriesForecast.xml'
) ForecastedRevenue
```

Example 4

This example includes the runtime-binded option where *numPeriods=%1*.

```
SELECT
"X - Airlines Traffic"."Month"."Dep Month" s_1,
"X - Airlines Traffic"."Passengers"."Passengers" s_2,
IFNULL(FORECAST(0.9*"X - Airlines Traffic"."Passengers"."Passengers" Target,
("X - Airlines Traffic"."Month"."Dep Month" Time_Month),'forecast',
```

```
'numPeriods=%1;predictionInterval=90',6),"X - Airlines
Traffic"."Passengers"."Passengers") s_3
FROM "X - Airlines Traffic"
```

Example 5

This example selects revenue forecast by year and month with a floating time grain.

```
SELECT "A - Sample Sales"."Time"."T05 Per Name Year" TimeYear1,
"A - Sample Sales"."Time"."T02 Per Name Month" Month1,
"A - Sample Sales"."Base Facts"."1- Revenue" Target1,FORECAST(
"A - Sample Sales"."Base Facts"."1- Revenue" ,
(), 'Forecast', 'numPeriods=%1;predictionInterval=70;',4) ForecastedRevenue
FROM "A - Sample Sales" order by Month1;
```

Example 6

This example selects revenue forecast by year and month with a floating time grain separately for each Region and Country Name.

```
SELECT
"A - Sample Sales"."Time"."T05 Per Name Year" TimeYear1,
"A - Sample Sales"."Time"."T02 Per Name Month" Month1,
"A - Sample Sales"."Base Facts"."1- Revenue" Target1,
"A - Sample Sales"."Office Regions"."D50 Region" Region,
"A - Sample Sales"."Office Regions"."D52 Country Name" Con,
FORECAST("A - Sample Sales"."Base Facts"."1- Revenue" ,
(),'Forecast','numPeriods=%1;predictionInterval=70;',4) ForecastedRevenue
FROM "A - Sample Sales"
order by region, con, Month1;
```

CLUSTER Function

The CLUSTER function arranges a set of records into groups based on one or more input expressions using K-Means or Hierarchical Clustering.

Options

This section contains information about the options and the option values you can use with the CLUSTER function.

Option Name	Description	Values
algorithm	The algorithm to use for clustering.	k-mean h-clustering
method	The method in the algorithm.	The methods for k-means algorithm are: Hartigan-Wong, Lloyd, Forgy, MacQueen. The methods for h-clustering algorithms are: ward.D, ward.D2, single, complete, average, mcquitty, median, or centroid.

Option Name	Description	Values
numClusters	The number of clusters. Every record is assigned to one of the clusters.	Integer
attributeNames	The attributes to consider for clustering.	arg1,arg2,arg3,arg4,arg5,arg6,arg7,arg8,arg9,arg10
maxIter	The maximum number of iterations.	Integer
normalizedDist	Normalizes the distance between 0 to 100 if set to TRUE.	True False
useRandomSeed	Set to TRUE by default. If set to TRUE, then is the value to use in PRODUCTION environment. If set to FALSE, then use <code>set.seed(initialSeed)</code> to ensure reproducibility, used in QA/ Debug Environment.	True False
initialSeed	Default is 250. This value is used only when <code>useRandomSeed</code> is set to FALSE.	Integer
clusterNamePrefix	Default is empty. If set, then this is the prefix for the cluster name.	Varchar
clusterNameSuffix	Default is empty. If set, then this is the suffix for the cluster name.	Varchar

Output Columns

The `CLUSTER` function output contains the following columns:

- **clusterID**
This column is the cluster number or ID.
- **clusterName**
This column is the name of the cluster. It is the same as **clusterID**.
- **clusterDescription**
The description can be added by the end user after the cluster dataset is persisted into DSS.

- **clusterSize**

This column is the number of elements in the current cluster.

- **distanceFromCenter**

This column indicates how far the current cluster element is from the center of the current cluster.

- **centers**

This column indicates the center of the current cluster.

Syntax

```
CLUSTER((dimension_expr1 , ... dimension_exprN), (expr1, .. exprN),  
output_column_name, options, [runtime_binded_options])
```

Where:

`dimension_expr` indicates a list of dimensions to cluster.

`expr` indicates a list of dimension attributes or measures to use to cluster `dimension_expr`.

`output_column_name` indicates the output column name for the cluster. Valid values are *clusterId*, *clusterName*, *clusterDescription*, *clusterSize*, *distanceFromCenter*, and *centers*.

`options` indicates a string list of name/value pairs separated by a semi-colon(;). You can include %1 ... %N that you can specify using `runtime_binded_options`.

`runtime_binded_options` indicates a comma separated list (,) of run-time binded columns or literal expressions.

Example 1

(Optional) Enter an example to illustrate your reference here.

```
SELECT "A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,  
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,  
CLUSTER(  
("A - Sample Sales"."Customers"."C0 Customer Number"),  
("A - Sample Sales"."Datamining"."M02- Expected Rev"),  
'clusterId', 'algorithm=k-means;numClusters=5;maxIter=10') ClusterId  
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number"  
< 1000;
```

Example 2

This example clusters Customer Number by Expected Revenue and Customer Age, and select the clusterID of every Customer Number less than 1000.

```
SELECT  
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,  
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,  
CLUSTER(  
("A - Sample Sales"."Customers"."C0 Customer Number"),  
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample  
Sales"."Customers"."C21 Customer Age"),  
'clusterId',
```

```
'algorithm=k-means;numClusters=5;maxIter=10'
) ClusterId
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <
1000;
```

Example 3

This example uses a custom cluster script XML. Note that the custom cluster script XML must reside in the same directory as other Advanced Analytics XML files.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
CLUSTER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'clusterId',
'algorithm=k-means;numClusters=5;maxIter=10;Script=filerepo://
CustomClusterScript.xml'
) ClusterId
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <
1000;
```

Example 4

This example shows runtime binded parameters.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
CLUSTER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'clusterId',
'algorithm=%2;numClusters=%1;maxIter=%3', 5, 'k-means', 10
) ClusterId
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <
1000;
```

Example 5

This example clusters Customer Number by Expected Revenue and Customer Age, and selects the clusterID and distanceFromCenter of each Customer Number less than 1000.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
CLUSTER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),'clusterId',
'algorithm=%2;numClusters=%1;maxIter=%3', 5,'k-means', 10
) ClusterId,
CLUSTER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
```

```

Sales"."Customers"."C21 Customer Age"),'distanceFromCenter',
'algorithm=%3;numClusters=%1;maxIter=%2', 5, 10, 'k-means'
) distanceFromCenter
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <
1000;

```

Example 6

This example clusters Customer Number by Expected Revenue and Customer Age, but separately for Brand and Product Type, and selects the clusterID and distanceFromCenter of each Customer Number less than 1000.

```

SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Products"."P2 Product Type" ProductType,
CLUSTER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'clusterName',
'algorithm=k-means;numClusters=%1;maxIter=%2;useRandomSeed=FALSE;', 2, 10
) clusterName
FROM "A - Sample Sales"
where Brand = 'FunPod' order by Brand, ProductType;

```

Example 7

This example clusters multiple columns (Product and Company) in the ID list. There is no partition dimension as both the ID list columns are in the projection list.

```

SELECT
"A - Sample Sales"."Products"."P1 Product" s_1,
"A - Sample Sales"."Offices"."D4 Company" s_2,
CLUSTER(
("A - Sample Sales"."Products"."P1 Product", "A - Sample Sales"."Offices"."D4
Company"),
("A - Sample Sales"."Base Facts"."2- Billed Quantity", "A - Sample Sales"."Base
Facts"."1- Revenue"),
'clusterName',
'algorithm=k-means;numClusters=%1;maxIter=%2;useRandomSeed=FALSE;', 5, 10) s_3,
"A - Sample Sales"."Base Facts"."2- Billed Quantity" s_4,
"A - Sample Sales"."Base Facts"."1- Revenue" s_5
FROM "A - Sample Sales";

```

Example 8

This example clusters multiple columns (Product and Company) in the ID list. There is one partition dimension (City).

```

SELECT
"A - Sample Sales"."Products"."P1 Product" s_1,
"A - Sample Sales"."Offices"."D4 Company" s_2,
"A - Sample Sales"."Office Regions"."D55 City" s_3,
CLUSTER(
("A - Sample Sales"."Products"."P1 Product", "A - Sample Sales"."Offices"."D4
Company"),
("A - Sample Sales"."Base Facts"."2- Billed Quantity", "A - Sample Sales"."Base

```

```

Facts"."1- Revenue"),
'clusterName',
'algorithm=k-means;numClusters=%1;maxIter=%2;useRandomSeed=FALSE;', 5, 10) s_4,
"A - Sample Sales"."Base Facts"."2- Billed Quantity" s_5,
"A - Sample Sales"."Base Facts"."1- Revenue" s_6
FROM "A - Sample Sales" order by s_3

```

OUTLIER Function

The OUTLIER function classifies a record as an outlier based on one or more input expressions using K-Means or Hierarchical Clustering or Multi-Variate Outlier detection Algorithms.

Options

The table contains information about the OUTLIER function options.

Option Name	Description	Values
attributeName	The attributes to consider for outlier detection.	arg1,arg2,arg3,arg4,arg5,arg6, arg7,arg8,arg9,arg10
algorithm	The algorithm to use.	mvoutlier (default) h-clustering k-means
useRandomSeed	This is TRUE by default. This is the value to use in PRODUCTION environment. If set to FALSE, then use <code>set.seed(initialSeed)</code> to ensure reproducibility. Used in QA/Debug environment.	TRUE FALSE
initialSeed	The attributes to consider for clustering.	Integer
topN	The maximum number of iterations. The default is set to 3. If <code>isTopNAsPercentage</code> is TRUE then return the topN % as outliers. Or else return the topN values as outliers.	Double
isTopNAsPercentage	The default is FALSE. This value is used with <code>topN</code> .	True False

Output Columns

The OUTLIER function output contains the following columns:

- **isOutlier**

This column uses True or False to indicate if the current element is an outlier.

- **distance**

This column indicates the distance from the center.

Syntax

```
OUTLIER((dimension_expr1 , ... dimension_exprN), (expr1, .. exprN),  
output_column_name, options, [runtime_binded_options]))
```

Where:

`dimension_expr` indicates a list of dimensions.

`expr` represents a list of dimension attributes or measures to find outlier.

`output_column_name` indicates the output column name. Valid values are *isOutlier* and *distance*.

`options` indicates a string list of name/value pairs separated by a semi-colon (;). The value can include %1 ... %N that you can specify using `runtime_binded_options`.

`runtime_binded_options` is a comma (,) separated list of run-time binded columns and or literal expressions.

Outliers Using mvoutlier Example

This example determines outliers for Customer Number based on Expected Revenue and using the mvoutlier algorithm.

```
SELECT  
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,  
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,  
OUTLIER(  
("A - Sample Sales"."Customers"."C0 Customer Number"),  
("A - Sample Sales"."Datamining"."M02- Expected Rev")  
'isOutlier',  
'algorithm=mvoutlier'  
) IsOutlier  
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <  
1000;
```

Outliers Using k-means Example

This example determines outliers for Customer Number based on Expected Revenue and Customer Age, and using algorithm k-means.

```
SELECT  
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,  
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,  
OUTLIER(  
("A - Sample Sales"."Customers"."C0 Customer Number"),  
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample  
Sales"."Customers"."C21 Customer Age"),  
'isOutlier',  
'algorithm=k-means'  
) IsOutlier  
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <  
1000;
```

Outliers Using a Custom Script Example

This example determines outliers using a custom outlier script XML. Note that the custom outlier script XML must reside in the same directory as other Advanced Analytics XML files.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID, "A - Sample
Sales"."Datamining"."M02- Expected Rev" ActualRevenue, OUTLIER( ( "A - Sample
Sales"."Customers"."C0 Customer Number"), ("A - Sample Sales"."Datamining"."M02-
Expected Rev", "A - Sample Sales"."Customers"."C21 Customer Age"), 'isOutlier',
'algorithm=mvoutlier;Script=filerepo://CustomOutlierScript.xml' ) IsOutlier FROM "A
- Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" < 1000;
```

Outliers Using Runtime Parameters Example

This example determines outliers using runtime binded parameters.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
OUTLIER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'isOutlier',
'algorithm=%1', 'mvoutlier'
) IsOutlier
FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers"."C0 Customer Number" <
1000;
```

Outliers and Distance for Customer Number Example

This example determines outliers and distance for Customer Number based on Expected Revenue and Customer Age.

```
SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustomerID,
"A - Sample Sales"."Datamining"."M02- Expected Rev" ActualRevenue,
OUTLIER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'isOutlier',
'algorithm=%1', 'mvoutlier'
) IsOutlier,
OUTLIER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Customers"."C21 Customer Age"),
'distance', 'algorithm=%1', 'mvoutlier')
Distance FROM "A - Sample Sales" WHERE "A - Sample Sales"."Customers".
"C0 Customer Number" < 1000;
```

Outliers for Customer Number Example

This example determines outliers for Customer Number based on Expected Revenue, Dyn Expected Rev, and Credit Rate, separately for each Brand and Office.

```

SELECT
"A - Sample Sales"."Customers"."C0 Customer Number" CustNo,
"A - Sample Sales"."Datamining"."M02- Expected Rev" Revenue,
"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Offices"."D1 Office" Office,
OUTLIER(
("A - Sample Sales"."Customers"."C0 Customer Number"),
("A - Sample Sales"."Datamining"."M02- Expected Rev", "A - Sample
Sales"."Datamining"."M03- Dyn Expected Rev", "A - Sample Sales".
"Customers"."C6 Credit Rate"), 'isOutlier', 'algorithm=k-means;'
) IsOutlier
FROM "A - Sample Sales" order by Brand, Office;

```

Multiple Columns ID List Example

This example includes multiple columns, Product and Company, in the ID list. There is no partition dimension as both ID list columns are in the projection list.

```

SELECT
"A - Sample Sales"."Products"."P1 Product" s_1,
"A - Sample Sales"."Offices"."D4 Company" s_2,
OUTLIER(
("A - Sample Sales"."Products"."P1 Product", "A - Sample Sales"."Offices".
"D4 Company"),
("A - Sample Sales"."Base Facts"."2- Billed Quantity", "A - Sample Sales".
"Base Facts"."1- Revenue"),
'isOutlier', 'algorithm=k-means;useRandomSeed=FALSE') s_3,
"A - Sample Sales"."Base Facts"."2- Billed Quantity" s_4,
"A - Sample Sales"."Base Facts"."1- Revenue" s_5
FROM "A - Sample Sales";

```

Multiple Columns City Example

This example includes multiple columns, Product and Company, in the ID list. The partition dimension is City.

```

SELECT
"A - Sample Sales"."Products"."P1 Product" s_1,
"A - Sample Sales"."Offices"."D4 Company" s_2,
"A - Sample Sales"."Office Regions"."D55 City" s_3,
OUTLIER(
("A - Sample Sales"."Products"."P1 Product", "A - Sample Sales"."Offices".
"D4 Company"),
("A - Sample Sales"."Base Facts"."2- Billed Quantity", "A - Sample Sales".
"Base Facts"."1- Revenue"),
'isOutlier', 'algorithm=k-means;useRandomSeed=FALSE') s_4,
"A - Sample Sales"."Base Facts"."2- Billed Quantity" s_5,
"A - Sample Sales"."Base Facts"."1- Revenue" s_6
FROM "A - Sample Sales" order by s_3;

```

REGR Function

The REGR function fits a linear model and returns the fitted values or model.

You can use this function to fit a linear curve on two measures.

Options

You can use the following options with the REGR function.

Option Name	Description	Values
algorithm	The algorithm to use for regression.	LM
attributeNames	The attribute to consider for regression.	arg1,arg2,arg3,arg4,arg5,arg6,arg7,arg8,arg9,arg10
targetNames	The number of iterations.	target
showModelDescription	This is set to TRUE by default. If set to TRUE, then the model description is written in JSON format.	TRUE FALSE

Syntax

```
REGR(y_axis_measure_expr, (x_axis_expr),
(category_expr1, ..., category_exprN), output_column_name,
options, [runtime_binded_options])
```

Where:

`y_axis_measure_expr` indicates the measure for which the regression model is to be computed.

`x_axis_measure_expr` indicates the measure used to determine the regression model for the `y_axis_measure_expr`.

`category_expr1, ..., category_exprN` indicates the dimension/dimension attributes used to determine the category for which the regression model for the `y_axis_measure_expr` is computed. One or up to five dimensions/dimension attributes can be provided as category columns.

`output_column_name` indicates the output column name for regression. Valid values are *fitted*, *intercept*, or *modelDescription*.

`options` indicates a string list of name/value pairs separated by a semi-colon (;). The value can include %1 ... %N, which can be specified using `runtime_binded_options`.

`runtime_binded_options` indicates an optional comma (,) separated list of runtime binded columns or literal expressions.

Example 1-3 Regression Model Based on Discount Amount Example

Regression Model Based on Discount Amount Example

This example selects fitted values for Revenue, determined using a regression model based on Discount amount for each Product Type and Brand, and partitioned on Region and Country Name.

```
SELECT
"A - Sample Sales"."Office Regions"."D50 Region" Region,
"A - Sample Sales"."Office Regions"."D52 Country Name" Con,
```

```

"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Products"."P2 Product Type" ProductType,
"A - Sample Sales"."Base Facts"."3- Discount Amount" Discount,
"A - Sample Sales"."Base Facts"."1- Revenue" ActualRevenue,
REGR(
"A - Sample Sales"."Base Facts"."1- Revenue" ,
("A - Sample Sales"."Base Facts"."3- Discount Amount"),
("A - Sample Sales"."Products"."P2 Product Type", "A - Sample Sales".
"Products"."P4 Brand"), 'FITTED', 'algorithm=%1', 'LM')
FROM "A - Sample Sales";

```

Example 1-4 Regression Model Revenue Example

Regression Model Revenue Example

This example selects fitted values for Revenue, determined using a regression model based on Discount amount for each Product Type, Brand, Region, and Country Name.

```

SELECT
"A - Sample Sales"."Office Regions"."D50 Region" Region,
"A - Sample Sales"."Office Regions"."D52 Country Name" Con,
"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Products"."P2 Product Type" ProductType,
"A - Sample Sales"."Base Facts"."3- Discount Amount" Discount,
"A - Sample Sales"."Base Facts"."1- Revenue" ActualRevenue,
REGR(
"A - Sample Sales"."Base Facts"."1- Revenue" ,
("A - Sample Sales"."Base Facts"."3- Discount Amount"), (),
'FITTED', 'algorithm=%1', 'LM')
FROM "A - Sample Sales";

```

Example 1-5 Regression Model Based on Discount Amount Example

Regression Model Based on Discount Amount Example

This example selects fitted values for Revenue, determined using a regression model based on Discount amount for each Product Type and Brand.

```

SELECT
"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Products"."P2 Product Type" ProductType,
"A - Sample Sales"."Base Facts"."3- Discount Amount" Discount,
"A - Sample Sales"."Base Facts"."1- Revenue" ActualRevenue,
REGR(
"A - Sample Sales"."Base Facts"."1- Revenue" ,
("A - Sample Sales"."Base Facts"."3- Discount Amount"),
("A - Sample Sales"."Products"."P2 Product Type", "A - Sample Sales".
"Products"."P4 Brand"), 'FITTED', 'algorithm=%1', 'LM')
FROM "A - Sample Sales";

```

Example 1-6 Regression Model Based Brand Example

Regression Model Based Brand Example

This example selects fitted values for Revenue, determined using a regression model based on Discount amount for each Product Type and Brand.

```

SELECT

```

```

"A - Sample Sales"."Products"."P4 Brand" Brand,
"A - Sample Sales"."Products"."P2 Product Type" ProductType,
"A - Sample Sales"."Base Facts"."3- Discount Amount" Discount,
"A - Sample Sales"."Base Facts"."1- Revenue" ActualRevenue,
REGR(
"A - Sample Sales"."Base Facts"."1- Revenue" ,
("A - Sample Sales"."Base Facts"."3- Discount Amount"),
(), 'FITTED', 'algorithm=%1', 'LM')
FROM "A - Sample Sales";

```

Oracle® Fusion Middleware Logical SQL Reference Guide for Oracle Business Intelligence Enterprise Edition, Release 12c
E71870-03

Copyright © 2016, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.