COMMERCE

MDEX Engine Analytics Guide

Version 6.5.2
October 2015

MDEX Engine Analytics Guide

Product version: 6.5.2

Release date: 10-22-15

Table of Contents

List of Examples

Preface

Oracle Commerce Guided Search is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Guided Search enables businesses to influence customers in each step of their search experience. At the core of Guided Search is the MDEX Engine™, a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Oracle Commerce Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. The Oracle Commerce Assembler dynamically assembles content from any resource and seamlessly combines it into results that can be rendered for display.

Oracle Commerce Experience Manager enables non-technical users to create, manage, and deliver targeted, relevant content to customers. With Experience Manager, you can combine unlimited variations of virtual product and customer data into personalized assortments of relevant products, promotions, and other content and display it to buyers in response to any search or facet refinement. Out-of-the-box templates and experience cartridges are provided for the most common use cases; technical teams can also use a software developer's kit to create custom cartridges.

About this guide

This guide describes how to add Endeca Analytics features to an Oracle Commerce application.

It assumes that you have read the *Oracle Commerce Getting Started Guide* and are familiar with terminology and basic concepts.

Who should use this guide

This guide is intended for developers who are building applications using Oracle Commerce with Analytics.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Oracle Support

Oracle Support provides registered users with answers to implementation questions, product and solution help, and important news and updates about Guided Search software.

You can contact Oracle Support through the My Oracle Support site at https://support.oracle.com.

1 Introduction to Endeca Analytics

This section introduces Endeca Analytics.

Related links

What is Endeca Analytics?

Analytics builds on the core capabilities of the MDEX Engine to enable applications that examine aggregate information such as trends, statistics, analytical visualizations, comparisons, and so on, all within the Guided Navigation interface.

This guide describes the key extensions to the MDEX Engine associated with Analytics:

- **Analytics API** A cornerstone feature of Analytics, the Analytics API extends the Presentation API to enable interactive applications that allow users to explore aggregate and statistical views of large databases using a Guided Navigation interface.
- **Charting API** These extensions to the Presentation API support graphical visualizations of Analytics results.
- **Date and time data** Additional data types now supported by the MDEX Engine enable applications to work with temporal data, performing time-based sorting, filtering, and analysis.
- **Key properties** Support for property- and dimension-level metadata allows customized application behavior such as automatic presentation of analytic measures and UI-level presentation of meta-information such as units, definitions, and data sources.

Where to find more information

This guide assumes familiarity with the core MDEX Engine platform, especially the Presentation API, Developer Studio, and the Information Transformation Layer.

Additional extensions to the core Oracle Commerce associated with Analytics are discussed elsewhere. In particular, support for normalized data is covered in the *Platform Services Forge Guide*.

2 The Analytics API

This section describes the Analytics API, which extends the Endeca Presentation API to enable applications that allow users to explore aggregate and statistical information.

Related links

What is the Analytics API?

To the base Presentation API provided by the core MDEX Engine, Endeca Analytics adds a powerful integrated Analytics API, which can be used to construct interactive analytics applications.

Some of the important features of the Endeca Analytics API are:

Tight integration with search and navigation The Endeca Analytics API allows analytical visualizations that update dynamically as the user refines the current search and navigation query, allowing Endeca Analytics to be controlled using the Oracle Commerce Guided Navigation interface. Further, analytics results support click-through to underlying record details, enabling end users to refine their navigation state directly from a view of their analytics sub-query results, exploring the details behind any statistic using the Guided Navigation interface.

Rich analytical functionality The Endeca Analytics API supports the computation of a rich set of analytics on records in an MDEX Engine, and in particular on the results of navigation, search, and other analytics operations. The API includes support for the following:

- Aggregation functions.
- Numeric functions.

- Composite expressions to construct complex derived functions.
- Grouped aggregations such as cross-tabulated totals over one or more dimensions.
- Top-k according to an arbitrary function.
- Cross-grouping comparisons such as time period comparisons.
- Intra-aggregate comparisons such as computation of the percentage contribution of one region of the data to a broader subtotal.
- Rich compositions of these features.

Efficiency Although the Endeca Analytics API allows the expression of a rich set of analytics, its functionality is constrained to allow efficient internal implementation, avoiding multiple table scans, complex joins, and so on. Good performance for analytics operations is essential for enabling the interactive response time associated with the Guided Navigation interface.

Familiarity The Endeca Analytics API uses concepts, structure, and terminology that are familiar to developers with a knowledge of SQL. API terminology, operators, and behavior match SQL for the majority of the Analytics API. Also, the Analytics API reuses familiar Presentation API classes and concepts, allowing developers to seamlessly move between working on navigation, search, and analytics functionality.

Basics of the Analytics API

The rest of this section covers the important features of the Analytics API, looks at sample analytics queries, and provides a detailed syntax reference.

A full reference of the Analytics API is not presented here, but is available in the *Presentation API for Java Reference (Javadoc)* and in the *Presentation and Logging API for .NET Reference (HTML Help)*.

Query input and embedding

The Endeca Analytics API provides the ability to request an arbitrary number of analytics operations based on the results of a single Oracle Commerce Navigation query. In other words, Endeca Analytics queries are embedded as sub-queries within a containing Oracle Commerce Navigation query.

This capability builds upon the Endeca API principle of "one page = one query", allowing applications to avoid costly round-trip requests to the MDEX Engine, and reduce overall page rendering time.

This embedding approach also supports the tight integration of Endeca search, navigation, and analytics: each analytics sub-query operates on the result records produced by the containing navigation query, allowing the corresponding analytics sub-query to update dynamically as the user refines the current search and navigation state.

The Endeca Analytics API provides two interfaces:

- An object-based programmatic interface
- A text-based syntax

The programmatic interface

As an extension of the Presentation API, the Endeca Analytics API provides a full, structured object-based programmatic interface (supported for Java and .NET APIs).

This standard object-level interface to Analytics queries is expected to be used in most production application settings. Like the ENEQuery interface, the Analytics API provides convenient methods for programmatic manipulation of the query, allowing the application to expose a broad set of GUI-level manipulators for analytical sub-queries.

Note

Code examples are provided in Java, but are easily converted into C# equivalents. All class names are equivalent. Accessor methods, method casing, and container/iterator (enumerator) syntax differ as appropriate for C# conventions. For details, see the Presentation and Logging API for .NET Reference (the HTML Help).

Example 2.1. Simple Analytics example

As a simple example, the following code snippet creates an empty Analytics query, and embeds it in a containing ENEQuery object:

```
ENEQuery query = new ENEQuery();
AnalyticsQuery analytics = new AnalyticsQuery();
query.setAnalyticsQuery(analytics);
```

The text-based syntax

The full programmatic interface may be inconvenient for ad-hoc query entry during debugging or for advanced application users. To satisfy these use cases, the Analytics API also provides a text-based syntax.

The majority of this syntax is based on a subset of the SQL language, providing familiarity for developers used to working with relational database systems. To create Analytics query objects based on the text-based syntax, a factory parser method is provided by the AnalyticsQuery class.

For example:

```
String str = … // Initialized to valid query
AnalyticsQuery analytics = AnalyticsQuery.parseQuery(str);
```

The competing desires of familiarity and efficiency are balanced by using a subset of SQL with additional enhancements that can be efficiently implemented.

Statements

Although Oracle Commerce Navigation requests can contain just a single AnalyticsQuery object, the AnalyticsQuery object can consist of an arbitrary number of Analytics statements, each of which represents a request to compute a set of Analytics result records.

An Analytics query can consist of any number of statements, each of which might compute related or independent analytics results.

Each Analytics statement in a query must be given a unique name, as these names are later used to retrieve Analytics results from the Navigation object returned by the MDEX Engine.

Example 2.2. Examples

The following code example creates two Analytics statements and inserts them into a containing AnalyticsQuery object:

```
AnalyticsQuery analytics = ...
Statement s1 = new Statement();
s1.setName("Statement One");
  // …populate s1 with analytics operations
analytics.add(s1);
Statement s2 = new Statement();
s2.setName("Statement Two");
  // …populate s2 with analytics operations
analytics.add(s2);
```

The same example using the text-based syntax is:

```
RETURN "Statement One" AS ...
RETURN "Statement Two" AS ...
```

The names assigned to Analytics statements are used to retrieve results from the Navigation object returned for the containing ENEQuery, as in the following example:

```
ENEQueryResults results = ...
AnalyticsStatementResult s1Results;
s1Results = results.getNavigation().
  getAnalyticsStatementResult("Statement One");
```

By default, the result of each statement in an Analytics query is returned to the calling application. But in some cases, an Analytics statement is only intended as input for other Analytics statements in the same query, not for presentation to the end user. In such cases, it is valuable to inform the system that the statement results are not needed, but are only defined as an intermediate step. Using the programmatic API, a Statement method is provided to accomplish this:

```
s1.setShown(false);
```

In the text-based syntax, this is accomplished using the DEFINE keyword in place of the RETURN keyword, for example:

```
DEFINE "Statement One" AS ...
```

The concept of query layering and inter-query references is described in later sections of this guide. So far, we have discussed statements in the abstract as parts of queries that compute analytics results, but have not described their capabilities or contents.

Aggregation/GROUP BY

The most basic type of statement provided by the Endeca Analytics API is the aggregation operation with GROUP BY, which buckets a set of Endeca records into a resulting set of aggregated Oracle Commerce Guide Search records.

In most Analytics applications, all Analytics statements are aggregation operations.

To define the set of resulting buckets for an aggregation operation, the operation must specify a set of GROUP BY dimensions and/or properties. The cross product of all values in these grouping dimensions and/or properties defines the set of candidate buckets. After associating input records with buckets, the results are automatically pruned to include only non-empty buckets. Aggregation operations correspond closely to the SQL GROUP BY concept, and the text-based Analytics syntax re-uses SQL keywords.

Example 2.3. Examples of GROUP BY

For example, suppose we have sales transaction data with records consisting of the following fields (properties or dimensions):

```
{ TransId, ProductType, Amount, Year, Quarter, Region,
  SalesRep, Customer }
```

such as:

```
{ TransId = 1, ProductType = "Widget", Amount = 100.00,
  Year = 2009, Quarter = "09Q1", Region  = "East",
  SalesRep = "J. Smith", Customer = "Customer1" }
```

If an Analytics statement uses "Region" and "Year" as GROUP BY dimensions, the statement results contain an aggregated Oracle Commerce Guide Search record for each valid, non-empty "Region" and "Year" combination. In the text-based syntax, this example would be expressed as:

```
DEFINE RegionsByYear AS
GROUP BY Region, Year
```

resulting in the aggregates of the form { Region, Year }, for example:

```
{ "East", "2008" }
{ "West", "2009" }
{ "East", "2009" }
```

The following Java code represents the equivalent operation using the programmatic API:

```
Statement stmnt = new Statement();
stmnt.setName("RegionsByYear");
GroupByList g = new GroupByList();
g.add(new GroupBy("Region"));
g.add(new GroupBy("Year"));
stmnt.setGroupByList(g);
```

The above example performs simple leaf-level group-by operations. It is also possible to group by a specified depth of each dimension. For example, if the "Region" dimension in the above example contained hierarchy such as Country, State, City, and the grouping was desired at the State level (one level below the root of the dimension hierarchy), the following Java syntax would be used:

```
g.add(new GroupBy("Region",1)); // Depth 1 is State
```

or in text form:

```
GROUP BY "Region":1
```

GROUP BY as a result of a computation

A GROUP BY key can be the result of a computation (the output of a SELECT expression), as long as that expression itself does not contain an aggregation function.

For example, the following syntax represents a correct usage of GROUP BY:

```
SELECT COALESCE(Person, 'Unknown Person')
as Person2, ... GROUP BY Person2
```

The following syntax is incorrect and results in an error (because Sales2 contains an aggregation function):

```
SELECT SUM(Sales) as Sales2, ... GROUP
BY Sales2
```

Specifying only Group

You can also use a GROUP statement to aggregate results into a single bucket.

For example, if you would like to use the SUM statement to return a single sum across a set of records, write the following query:

```
RETURN "ReviewCount" AS SELECT
SUM(number_of_reviews) AS "NumReviews"
GROUP
```

This query returns one record for the property `"NumReviews"` where the value is the sum over the property `"number_of_reviews"`.

Expressions/SELECT AS

This topic describes SELECT AS operations, contains lists of Analytics functions (aggregation, numeric and time/date), and describes the COALESCE expression.

Having created a set of aggregates using a GROUP BY operation, it is typical for Analytics queries to compute one or more derived analytics in each resulting bucket. This is accomplished using SELECT AS operations and Analytics expressions.

Each aggregation operation can declare an arbitrary set of named expressions, sometimes referred to as derived properties, using SELECT AS syntax. These expressions represent aggregate analytic functions that are computed for each aggregated Oracle Commerce Guide Search record in the statement result. Expressions can make use of a broad array of operators, which are described in the following section.

List of aggregation functions

Analytics supports the following aggregation functions:

Function	Description
AVG	Computes the arithmetic mean value for a field.

Function	Description
COUNT	Counts the number of records with valid non-null values in a field for each GROUP BY result.
COUNTDISTINCT	Counts the number of unique, valid non-null values in a field for each GROUP BY result.
MAX	Finds the maximum value for a field.
MIN	Finds the minimum value for a field.
MEDIAN	Finds the median value for a field.
STDDEV	Computes the standard deviation for a field.
ARB	Selects an arbitrary but consistent value from the set of values in a field.
SUM	Computes the sum of field values.
VARIANCE	Computes the variance (that is, the square of the standard deviation) for a field.

List of numeric functions

Analytics supports the following numeric functions:

Function	Description
addition	The addition operator (+).
subtraction	The subtraction operator (-).
multiplication	The multiplication operator (*).
division	The division operator (/).
ABS	Returns the absolute value of n. If n is 0 or a positive integer, returns n; otherwise, n is multiplied by -1.
CEIL	Returns the smallest integer value not less than n.
EXP	Exponentiation, where the base is e. Returns the value of e (the base of natural logarithms) raised to the power of n.
FLOOR	Returns the largest integer value not greater than n.
LN	Natural logarithm. Computes the logarithm of its single argument, the base of which is e.
LOG	Logarithm. log(n, m) takes two arguments, where n is the base, and m is the value you are taking the logarithm of.

Function	Description
MOD	Modulo. Returns the remainder of n divided by m. Analytics uses the fmod floating point remainder, as defined in the C/POSIX standard.
ROUND	Returns a number rounded to the specified decimal place. The unary version drops the decimal (non-integral) portion of the input. The binary version allows you to set the number of spaces at which the number is rounded: • Positive second arguments specified to this function correspond to the number of places that must be returned after the decimal point. For example, `ROUND(123.4567, 3) = 123.457` • Negative second arguments correspond to the number of places that must be returned before the decimal point. For example, `ROUND(123.4567, -3) = 100.0`
SIGN	Returns the sign of the argument as -1, 0, or 1, depending on whether n is negative, zero, or positive.
SQRT	Returns the nonnegative square root of n.
TRUNC	Returns the number n, truncated to m decimals. If m is 0, the result has no decimal point or fractional part. The unary version drops the decimal (non-integral) portion of the input, while the binary version allows you to set the number of spaces at which the number is truncated.
SIN	The sine of n, where the angle of n is in radians.
COS	The cosine of n, where the angle of n is in radians.
TAN	The tangent of n, where the angle of n is in radians.
POWER	Returns the value of n raised to the power of m.
TO_DURATION	Casts an integer into a number of milliseconds so that it can be used as a duration. When the unary version of TO_DURATION is given a value of type double, it removes the decimal portion and converts the integer portion to a duration.

Time/date functions

Time/date functions such as EXTRACT and TRUNC are discussed in the section about temporal properties. These operators may be composed to construct arbitrary, complex derived expressions. As a simple example using the text-based syntax, we could compute the total number of deals executed by each sales representative as follows:

```
RETURN DealsPerRep AS
SELECT COUNT(TransId) AS NumDeals
GROUP BY SalesRep
```

Continuing the example from the "Aggregation/GROUP BY" topic, if we wanted each result aggregate to be assigned the ratio of the average "Amount" value to the maximum "Amount" value as the property "AvgOverMax", we would use the following Java API syntax:

```
ExprBinary e = new ExprBinary(ExprBinary.DIVIDE,
 new ExprAggregate(ExprAggregate.AVG,
  new ExprKey("Amount")),
 new ExprAggregate(ExprAggregate.MAX,
  new ExprKey("Amount")));
SelectList s = new SelectList();
s.add(new Select("AvgOverMax",e));
stmnt.setSelectList(s);
```

The same example in the text-based syntax is:

```
SELECT AVG(Amount) / MAX(Amount) AS AvgOverMax
```

COALESCE expression

The COALESCE expression allows for user-specified null-handling. You can use COALESCE to evaluate records for multiple values and return the first non-null value encountered, in the order specified. The following requirements apply:

- You can specify two or more arguments to COALESCE
- Arguments that you specify to COALESCE should all be of the same type
- The types of arguments that COALESCE takes are: integer, integer64, double, and string.
- COALESCE supports alphanumeric values, but for typing reasons it does not support mixing numeric values and alphanumeric ones.
- You cannot specify dimensions as arguments to COALESCE. However, if you have a dimension mapped to a property, you can then specify this property to COALESCE and this will result in a valid query.
- The COALESCE expression can only be used in a SELECT clause, and not in other clauses (such as WHERE)

In the following example, all records without a specified price are treated as zero in the computation:

```
AVG(COALESCE(price, 0))
```

COALESCE can also be used on its own, for example:

```
SELECT COALESCE(price, 0) WHERE …
```

Using the COUNT and COUNTDISTINCT functions

Review these examples before implementing the COUNT and COUNTDISTINCT functions.

The COUNT function counts the number of records with valid non-null values in a field for each GROUP BY result. For example, suppose there are four records with the value "small" in the Size field, and some of them have values in the Color field:

```
Record 1: Size=small, Color=red, Color=white
Record 2: Size=small, Color=blue, Color=green
Record 3: Size=small, Color=black
Record 4: Size=small
```

The query "RETURN result AS SELECT COUNT(Color) as Total GROUP BY Size" will return the result:

```
Record 1: Size=small, Total=3
```

The query returns "3", because in the "small" result group, three records had valid Color assignments.

Note

The COUNT function is counting **records** with valid assignments (three in this example), **not** the number of different values that appear in the field (five in this example: red, white, blue, green and black).

For single-assigned properties, the COUNTDISTINCT function returns the number of unique, valid non-null values in a field for each GROUP BY result. For example, suppose there are four records with the value "small" in the Size field and different single values in the Color field:

```
Record 1: Size=small, Color=red
Record 2: Size=small, Color=blue
Record 3: Size=small, Color=red
Record 4: Size=small
```

The query "RETURN result AS SELECT COUNTDISTINCT (Color) as Total GROUP BY Size" would return:

```
Record 1: Size=small, Total=2
```

The total is "2" because across all of the records in this group, there are two unique, valid, non-null values in the Color field: "red" and "blue."

Note

COUNTDISTINCT should never be used with multi-assigned properties. The results of the query may be misleading if records can have multiple values for one property (for example: **Record 1: Size=small, Color=red, Color=white, Color=blue**).

Nested queries/FROM

But using FROM syntax, an aggregation operation can specify that its input be obtained from the output of any previously declared statement.

The results of an Analytics statement consist of a set of aggregated records that can be used like any other Oracle Commerce Guide Search records. Because of this, aggregation operations can be layered upon one another. By default, the source of records for an Analytics statement is the result of the containing search and navigation query. But using FROM syntax, an aggregation operation can specify that its input be obtained from the output of any previously declared statement.

Example 2.4. Example of using FROM syntax

For example, one aggregation query might compute the total number of transactions grouped by "Quarter" and "Sales Rep". Then, a subsequent aggregation can group these results by "Quarter", computing the average number of transactions per "Sales Rep". This example can be expressed in the text-based syntax as:

```
DEFINE RepQuarters AS
SELECT COUNT(TransId) AS NumTrans
GROUP BY SalesRep, Quarter ;

RETURN Quarters AS
SELECT AVG(NumTrans) AS AvgTransPerRep
FROM RepQuarters
GROUP BY Quarter
```

RepQuarters produces aggregates of the form { SalesRep, Quarter, NumTrans }, such as:

```
{ J. Smith, 09Q1, 10 }
{ J. Smith, 09Q2, 3 }
{ F. Jackson, 08Q4, 10 }
...
```

and Quarters returns aggregates of the form { Quarter, AvgTransPerRep }, such as:

```
{ 083Q4, 10 }
{ 09Q1, 4.5 }
{ 09Q2, 6 }
...
```

The same example using the programmatic API is:

```
// First, define "SalesRep" "Quarter" buckets with
// "TransId" counts.
Statement repQuarters = new Statement();
repQuarters.setName("RepQuarters");
repQuarters.setShown(false);

GroupByList rqGroupBys = new GroupByList();
rqGroupBys.add(new GroupBy("SalesRep"));
rqGroupBys.add(new GroupBy("Quarter"));
repQuarters.setGroupByList(rqGroupBys);

Expr e = new ExprAggregate(ExprAggregate.COUNT,
                           new ExprKey("TransId"));
SelectList rqSelects = new SelectList();
rqSelects.add(new Select("NumTrans",e));
repQuarters.setSelectList(rqSelects);

// Now, feed these results into "Quarter" buckets
// computing averages
Statement quarters = new Statement();
quarters.setName("Quarters");
quarters.setFromStatementName("RepQuarters");
GroupByList qGroupBys = new GroupByList();
qGroupBys.add(new GroupBy("Quarter"));
quarters.setGroupByList(qGroupBys);

e = new ExprAggregate(ExprAggregate.AVG,
                      new ExprKey("NumProducts"));
SelectList qSelects = new SelectList();
qSelects.add(new Select("AvgTransPerRep", e));
quarters.setSelectList(qSelects);
```

Inter-statement references

Multiple analytics sub-queries can be specified within the context of a single Oracle Commerce Navigation query, each corresponding to a different analytical view, or to a sub-total at a different granularity level.

Statements can be nested within one another to compute layered Analytics.

Additionally, using a special cross-table referencing facility provided by the SELECT expression syntax, expressions can make use of values from other computed statements. This is often useful for when coarser subtotals are required for computing analytics within a finer-grained bucket.

For example, if computing the percent contribution for each sales representative last year, the overall year total is needed to compute the value for each individual rep. Queries such as this can be composed using inter-statement references.

Example 2.5. Examples of inter-statement references

As an example, suppose we want to compute the percentage of sales per "ProductType" per "Region". One aggregation computes totals grouped by "Region", and a subsequent aggregation computes totals grouped by "Region" and "ProductType". This second aggregation would use expressions that referred to the results from the "Region" aggregation. That is, it would allow each "Region" and "ProductType" pair to compute the percentage of the full "Region" subtotal represented by the "ProductType" in this "Region".

The first statement computes the total product sales for each region:

```
DEFINE RegionTotals AS
SELECT SUM(Amount) AS Total
GROUP BY Region
```

Then, a statement uses the "RegionTotals" results defined above to determine the percentage for each region, making use of the inter-statement reference syntax (square brackets for addressing and dot operator for field selection).

```
RETURN ProductPcts AS
SELECT
  100 * SUM(Amount) / RegionTotals[Region].Total AS PctTotal
GROUP BY Region, ProductType
```

The bracket operator specifies that we are referencing the result of RegionTotals statement whose group-by value is equal to the local ProductPcts bucket's value for the Region dimension. The dot operator indicates that we are referencing the Total field in the specified RegionTotals bucket.

The above example makes use of referencing values in a separate statement, but the reference operator can also be used to reference values within the current statement. This is useful for computing trends that change over time, such as year-on-year sales change, which could be expressed as:

```
RETURN YearOnYearChange AS
SELECT SUM(Amount) AS TotalSales,
  SUM(Amount) - YearOnYearChange[Year-1].TotalSales AS Change
GROUP BY Year
```

This same example expressed using the programmatic API is:

```
Statement stmnt = new Statement();
stmnt.setName("YearOnYearChange");
GroupByList groupBys = new GroupByList();
groupBys.add(new GroupBy("Year"));
stmnt.setGroupByList(groupBys);
SelectList selects = new SelectList();

Expr totalSales = new ExprAggregate(ExprAggregate.SUM,
                                    new ExprKey("Amount"));
selects.add(new Select("TotalSales",totalSales));

LookupList lookup = new LookupList();
lookup.add(new ExprBinary(ExprBinary.MINUS,
           new ExprKey("Year"), new ExprConstant("1")));
Expr change = new ExprBinary(ExprBinary.MINUS, totalSales,
    new ExprLookup("YearOnYearChange", "TotalSales", lookup));
selects.add(new Select("Change",change));
stmnt.setSelectList(selects);
```

Result ordering/ORDER BY

The order of result records returned by an aggregation operation can be controlled using ORDER BY operators.

Records can be ordered by any of their property, dimension, or derived values, ascending or descending, in any combination.

Example 2.6. Examples with ORDER BY operators

For example, to order "SaleRep" aggregates by their total "Amount" values, the following Java API calls would be used:

```
Statement reps = new Statement();
reps.setName("Reps");
GroupByList groupBys = new GroupByList();
groupBys.add(new GroupBy("SalesRep"));
reps.setGroupByList(groupBys);
Expr e = new ExprAggregate(ExprAggregate.SUM,
  new ExprKey("Amount"));
SelectList selects = new SelectList();
selects.add(new Select("Total",e));
reps.setSelectList(selects);
OrderByList o = new OrderByList();
o.add(new OrderBy("Total", false));
reps.setOrderByList(o);
```

The same example in the text-based syntax is:

```
DEFINE Reps AS
SELECT SUM(Amount) AS Total
GROUP BY SalesRep
ORDER BY Total DESC
```

Paging and rank filtering/PAGE

By default, any statement that returns results to the application returns all results. In some cases, it is useful to request results in smaller increments for presentation to the user (such as presenting the sales reps ten at a time, with links to page forward and backward).

For example, the following query groups the records by SalesRep, returning the 10th through 19th resulting buckets (order in this case is arbitrary but consistent between queries):

```
DEFINE Reps AS
GROUP BY SalesRep
PAGE(10,19)
```

Paging can also be used in combination with ORDER BY to achieve "top-k" type queries. For example, the following query returns the top 10 sales reps by total sales:

```
DEFINE Reps AS
```

```
SELECT SUM(Amount) AS Total
GROUP BY SalesRep
ORDER BY Total DESC
PAGE(0,10)
```

This same example using the programmatic API is:

```
Statement reps = new Statement();
reps.setName("Reps");
GroupByList groupBys = new GroupByList();
groupBys.add(new GroupBy("SalesRep"));
reps.setGroupByList(groupBys);
Expr e = new ExprAggregate(ExprAggregate.SUM,
  new ExprKey("Amount"));
SelectList selects = new SelectList();
selects.add(new Select("Total",e));
reps.setSelectList(selects);
OrderByList o = new OrderByList();
o.add(new OrderBy("Total", false));
reps.setOrderByList(o);
reps.setReturnRows(0,10); // start at 0th and return 10 records
```

Filters/WHERE, HAVING

The Analytics API supports a general set of filtering operations. As in SQL, these can be used to filter the input records considered by a given statement (WHERE) and the output records returned by that statement (HAVING).

You can also filter input records for each expression to compute analytic expressions for a subset of the result buckets.

A variety of filter operations are supported, such as numeric and string value comparison functions (such as equality, inequality, greater than, less than, between, and so on), and Boolean operators (AND, OR, or NOT) for creating complex filter constraints.

For example, we could limit the sales reps to those who generated at least $10,000:

```
RETURN Reps AS
SELECT SUM(Amount) AS SalesTotal
GROUP BY SalesRep
HAVING SalesTotal > 10000
```

You could further restrict this analytics sub-query to only the sales in the Western region:

```
RETURN Reps AS
SELECT SUM(Amount) AS SalesTotal
WHERE Region = 'West'
GROUP BY SalesRep
HAVING SalesTotal > 10000
```

Alternatively, you could use the `WHERE` filter with an ID of the dimension value based on which we are restricting the results. For example, if the ID of the **`"West"`** dimension is **`"1234"`**, the `WHERE` filter looks like this:

```
RETURN Reps AS
SELECT SUM(Amount) AS SalesTotal
WHERE Dval(1234)
GROUP BY SalesRep
HAVING SalesTotal > 10000
```

Note

Ensure that you specify a valid attribute to the WHERE filter (or to any other filter clause).

The example with the name of the dimension "West", as expressed using the programmatic API, is:

```
Statement reps = new Statement();
reps.setName("Reps");
reps.setWhereFilter(
  new FilterCompare("Region", FilterCompare.EQ, "West"));
reps.setHavingFilter(
  new FilterCompare("SalesTotal", FilterCompare.GT, "10000"));
GroupByList groupBys = new GroupByList();
groupBys.add(new GroupBy("SalesRep"));
reps.setGroupByList(groupBys);
Expr e = new ExprAggregate(ExprAggregate.SUM,
  new ExprKey("Amount"));
SelectList selects = new SelectList();
selects.add(new Select("SalesTotal",e));
reps.setSelectList(selects);
```

As mentioned above, filters can be specified for each expression. For example, a single query can include two expressions, where one expression computes the total for the entire aggregate while another expression computes the total for a particular sales representative:

```
RETURN QuarterTotals AS SELECT
  SUM(Amount) AS SalesTotal,
  SUM(Amount) WHERE SalesRep = 'John Smith' AS JohnTotal
GROUP BY Quarter
```

This would give us both the total overall sales and the total sales for John Smith in each quarter. The same example in the Java API is:

```
Statement stmnt = new Statement();
stmnt.setName("QuarterTotals");
GroupByList groupBys = new GroupByList();
groupBys.add(new GroupBy("Quarter"));
stmnt.setGroupByList(groupBys);

SelectList selects = new SelectList();
ExprAggregate e = new ExprAggregate(ExprAggregate.SUM,
  new ExprKey("Amount"));
selects.add(new Select("SalesTotal",e));

e = new ExprAggregate(ExprAggregate.SUM, new ExprKey("Amount"));
e.setFilter(
  new FilterCompare("SalesRep", FilterCompare.EQ,
   John Smith"));
```

```
selects.add(new Select("JohnTotal",e));

stmnt.setSelectList(selects);
```

When the HAVING clause is applied

In a query such as:

```
SELECT a AS b GROUP BY c HAVING d>7
```

the HAVING clause applies after the "a AS b" clause, so that it filters out the records returned by the SELECT, not the records on which the SELECT operates. If we were to write the query as a set of function calls, it would look like this:

```
HAVING(d>7,SELECT(a AS b,GROUP(BY c)))
```

That is, the SELECT gets data from the GROUP BY, does its calculations, and passes its results on to the HAVING to be filtered. If you are familiar with SQL, this may be a surprise, because, in SQL, HAVING has the opposite effect: it filters the results of the GROUP BY, not the results of the SELECT.

Analytics results

Each Analytics statement produces a set of virtual Oracle Commerce Guide Search records that can be used in application code like any other Oracle Commerce Guide Search records (ERec objects).

Results for a statement can be retrieved using the name for that statement. For example, if we used a query like:

```
RETURN "Product Totals" AS ...
```

Then we could retrieve the results as follows:

```
ENEQueryResults results = ...
AnalyticsStatementResult analyticsResults;
analyticsResults = results.getNavigation().
  getAnalyticsStatementResult("Product Totals");
```

The AnalyticsStatementResult object provides access to an iterator over the result records, for example:

```
Iterator recordIter = analyticsResults.getERecIter();
```

The AnalyticsStatementResult object also provides access to the total number of result records associated with the statement (which may be greater than the number returned if a PAGE operator was used).

The AnalyticsStatementResult object also provides access to an error message for the analytics statement. The error message, if not null, provides information about why the query could not be evaluated.

The MDEX Engine may evaluate some of the statements in an analytics request but not others if errors in the statements (for example, use of a non-existent group-by key) are independent.

In addition to per-statement error messages, the Navigation object provides access to a global error message for the complete analytics query. This can be non-null in cases where statement evaluation was not possible because of query-level problems, for example not all statements being assigned unique names.

Temporal Analytics

You can use Time, DateTime, and Duration properties for these Analytics operations: `TRUNC` and `EXTRACT`.

- `TRUNC` rounds a DateTime down to a coarser granularity bucket.
- `EXTRACT` extracts a portion of a DateTime, such as the day of the week.

In addition to these special-purpose functions, Date and Time values can support arithmetic operations if you cast double or integer fields to a Duration using `TO_DURATION`. For example:

- A Duration may be added to a Time or a DateTime to obtain a new Time or DateTime.
- Two Times or two DateTimes may be subtracted to obtain a Duration.
- Two Durations may be added or subtracted to obtain a new Duration.

TRUNC

The TRUNC function can be used to round a DateTime value down to a coarser granularity.

For example, this is useful when performing a GROUP BY on DateTime data based on coarser time ranges (such as GROUP BY "Quarter" given DateTime values). The syntax of the function is:

```
<TruncExpr>     ::=  TRUNC(<expr>,<DateTimeUnit>)
<DateTimeUnit>  ::=  SECOND | MINUTE | HOUR |
                     DATE | WEEK | MONTH | QUARTER | YEAR
```

For example, given a DateTime property "TimeStamp" with a value representing 10/13/2009 11:35:12.000, the TRUNC operator could be used to compute the following results (represented in pretty-printed form for clarity; actual results would use standard `DateTime` format):

```
TRUNC("TimeStamp", SECOND)  = 10/13/2009 11:35:12.000
TRUNC("TimeStamp", MINUTE)  = 10/13/2009 11:35:00.000
TRUNC("TimeStamp", HOUR)    = 10/13/2009 11:00:00.000
TRUNC("TimeStamp", DATE)    = 10/13/2009 00:00:00.000
TRUNC("TimeStamp", WEEK)    = 10/08/2009 00:00:00.000
TRUNC("TimeStamp", MONTH)   = 10/01/2009 00:00:00.000
TRUNC("TimeStamp", QUARTER) = 10/01/2009 00:00:00.000
TRUNC("TimeStamp", YEAR)    = 01/01/2009 00:00:00.000
```

As a simple example of using this functionality, the following query groups transaction records containing TimeStamp and Amount properties by quarter:

```
RETURN Quarters AS
SELECT SUM(Amount) AS Total,
       TRUNC(TimeStamp, QUARTER) AS Qtr
GROUP BY Qtr
```

EXTRACT

The EXTRACT function extracts a portion of a DateTime, such as the day of the week or month of the year.

This is useful in situations where the data must be filtered or grouped by a slice of its timestamps, for example computing the total sales that occurred on any Monday.

```
<ExtractExpr>   ::=  EXTRACT(<expr>,<DateTimeUnit>)
<DateTimeUnit>  ::=  SECOND | MINUTE | HOUR | DAY_OF_WEEK |
                     DAY_OF_MONTH | DAY_OF_YEAR | DATE | WEEK |
                     MONTH | QUARTER | YEAR
```

For example, given a DateTime property "TimeStamp" with a value representing 10/13/2009 11:35:12.000, the EXTRACT operator could be used to compute the following results:

```
EXTRACT("TimeStamp", SECOND)       = 12
EXTRACT("TimeStamp", MINUTE)       = 35
EXTRACT("TimeStamp", HOUR)         = 11
EXTRACT("TimeStamp", DATE)         = 13
EXTRACT("TimeStamp", WEEK)         = 40   /* Zero-indexed */
EXTRACT("TimeStamp", MONTH)        = 10
EXTRACT("TimeStamp", QUARTER)      = 3    /* Zero-indexed */
EXTRACT("TimeStamp", YEAR)         = 2009
EXTRACT("TimeStamp", DAY_OF_WEEK)  = 4    /* Zero-indexed */
EXTRACT("TimeStamp", DAY_OF_MONTH) = 13
EXTRACT("TimeStamp", DAY_OF_YEAR)  = 286  /* Zero-indexed */
```

As a simple example of using this functionality, the following query groups transaction records containing TimeStamp and Amount properties by quarter, and for each quarter computes the total sales that occurred on a Monday (DAY_OF_WEEK=1):

```
RETURN Quarters AS
SELECT SUM(Amount) AS Total
       TRUNC(TimeStamp, QUARTER) AS Qtr
WHERE EXTRACT(TimeStamp,DAY_OF_WEEK) = 1
GROUP BY Qtr
```

Sample queries

This section presents sample queries for a number of representative use cases.

Queries are presented in the text-based syntax for clarity. For the purposes of these examples, we assume that we have sales transaction data with records consisting of the following fields (properties or dimensions):

```
{ TransId, ProductType, Amount, Year, Quarter, Region,
  SalesRep, Customer }
```

such as:

```
{ TransId = 1, ProductType = "Widget", Amount = 100.00,
  Year = 2009, Quarter = "09Q1", Region  = "East",
  SalesRep = "J. Smith", Customer = "Customer1" }
```

Simple cross-tabulation

Compute total, average, and medial sales amounts, grouped by Quarter and Region.

This query represents a simple cross-tabulation of the raw data.

```
RETURN Results AS
SELECT
  SUM(Amount) AS Total,
  AVG(Amount) AS AvgDeal,
  MEDIAN(Amount) AS MedianDeal
GROUP BY Quarter, Region
```

This could also be expressed using multiple statements:

```
RETURN Totals AS
SELECT SUM(Amount) AS Total
GROUP BY Quarter, Region ;

RETURN Avgs AS
SELECT AVG(Amount) AS AvgDeal
GROUP BY Quarter, Region ;

RETURN Medians AS
SELECT MEDIAN(Amount) AS MedianDeal
GROUP BY Quarter, Region
```

These queries produce the same information in different structures. For example, the first is more useful if the application is generating a single table with the total, average, and median presented in each cell. The second is more convenient if the application is generating three tables, one each for total, average, and median.

In terms of efficiency, the queries are not significantly different (the second is optimized to avoid multiple table scans). But the second returns more data (the Quarter and Region groupings must be repeated for each result), and thus is marginally less efficient.

Top-k

Compute the best 10 SalesReps based on total sales from the first quarter of this year:

```
RETURN BestReps AS
SELECT SUM(Amount) AS Total
WHERE Quarter = '09Q1'
GROUP BY SalesRep
ORDER BY Total DESC
PAGE(0,10)
```

Subset comparison

Compute the percent of sales where ProductType="Widgets", grouped by Quarter and Region. This query combines totals on all elements of a grouping (total sales) with totals on a filtered set of elements (sales for "Widgets" only).

```
RETURN Results AS
SELECT
  (SUM(Amount) WHERE ProductType='Widgets') /
   SUM(Amount) AS PctWidgets
GROUP BY Quarter, Region
```

Nested aggregation

Compute the average number of transactions per sales representative grouped by Quarter and Region.

This query represents a multi-level aggregation. First, transactions must be grouped into sales reps to get per-rep transaction counts. Then, these rep counts must be aggregated into averages by quarter and region.

```
DEFINE DealCount AS
SELECT COUNT(TransId) AS NumDeals
GROUP BY SalesRep, Quarter, Region ;

RETURN AvgDeals AS
SELECT AVG(NumDeals) AS AvgDealsPerRep
FROM DealCount
GROUP BY Quarter, Region
```

Inter-aggregate references example

Compute year-to-year change in sales, grouped by Year and ProductType.

This query requires inter-aggregate use of information. To compute the change in sales for a given quarter, the total from the prior quarter and the current quarter must be referenced.

```
RETURN YearOnYearChange AS
SELECT SUM(Amount) AS TotalSales,
  SUM(Amount) -
```

```
    YearOnYearChange[Year-1,ProductType].TotalSales AS Change
GROUP BY Year, ProductType
```

Inter-statement references example

For each quarter, compute the percentage of sales due to each product type.

This query requires inter-statement use of information. To compute the sales of a given product as a percentage of total sales for a given quarter, the quarterly totals must be computed and stored so that calculations for quarter/product pairs can retrieve the corresponding quarterly total.

```
DEFINE QuarterTotals AS
SELECT SUM(Amount) AS Total
GROUP BY Quarter ;

RETURN ProductPcts AS
SELECT
  100 * SUM(Amount) / QuarterTotals[Quarter].Total AS PctTotal
GROUP BY Quarter, ProductType
```

Text-based syntax reference

The examples and discussion of the query syntax provided in previous sections give an overview of the API and its intended use. This section is a full reference for the text-based query syntax.

A full reference of the Analytics API is not presented here, but is available in the *Presentation API for Java (Javadoc)* and in the *Presentation and Logging API for .NET Reference (HTML Help)*.

Syntax (BNF)

The basic structure of a query is an ordered list of statements. The result of a query is a set of named record sets (one per statement).

```
<Query>        ::=  <Statements>
<Statements>   ::=  <Statement> [; <Statements>]
```

A statement defines a record set by specifying:

1. A name for the resulting record set.
2. What properties (or derived properties) its records should contain (SELECTs).
3. The input record set (FROM).
4. A filter to apply to that input (WHERE).

5. A way to group input records (a mapping from input records to output records, GROUP BY).

6. A filter to apply to output records (HAVING).

7. A way to order the output records (ORDER BY).

8. A way to page through the output records (return only a subset of the output, PAGE).

```
<Statement>    ::=  (DEFINE | RETURN) <Key> AS <Select>
                       [<From>]
                       [<Where>]
                       [<GroupBy>]
                       [<Having>]
                       [<OrderBy>]
                       [<Page>]
```

A statement includes a list of assignments that compute the derived properties populated into the resulting records:

```
<Select>       ::=  SELECT <Assigns>
<Assigns>      ::=  <Assign> [, <Assigns>]
<Assign>       ::=  <Expr> AS <Key>
```

A statement's input record set is either the result of a previous statement, the navigation state's records (NavStateRecords), or the Dgraph's base records (AllBaseRecords). The omission of the FROM clause implies FROM NavStateRecords.

```
<From>         ::=  FROM <Key>
               ::=  FROM NavStateRecords
               ::=  FROM AllBaseRecords
```

As in SQL, an optional WHERE clause represents a pre-filter on inbound records, and an optional HAVING clause represents a post-filter on the output records:

```
<Where>        ::=  WHERE <Filter>
<Having>       ::=  HAVING <Filter>
```

Input records need to be mapped to output records for the purposes of populating derived values in the output records. The input records provide the values used in expressions and Where filters.

Ways of mapping inputs to outputs

There are three ways to map input records to output (or aggregated) records.

- If a list of properties and/or dimensions is given, all input records containing identical values for those properties map to (group to) the same output record.
- If only GROUP is specified, all input records are mapped to a single output record.

- The absence of a GROUP BY clause is taken to mean that each input record is mapped to a unique output record.

```
<GroupBy>      ::=  GROUP
               ::=  GROUP BY <Groupings>
<Groupings>    ::=  <Grouping> [, <Groupings>]
<Grouping>     ::=  <Key>
               ::=  <Key>:<int>
```

Note

Colon operator requests grouping by dimension values at a specified depth.

If an input record contains multiple values corresponding to the same grouping key, the default behavior is that the record maps to all possible output records. An optional ORDER BY clause can be used to order the records in the resulting record set. An element in the ORDER BY clause specifies a property or dimension to sort (or break ties) by and in what direction: (growing values = ASCending; shrinking values = DESCending). The default order is ascending.

```
<OrderBy>      ::=  ORDER BY <OrderList>
<OrderList>    ::=  <Order> [, <OrderList>]
<Order>        ::=  <Key> [ASC | DESC]
```

The PAGE(i,n) operator allows the returned record set to be limited to n records starting with the record at index i. The 'Page' clause can be used with or without a preceding ORDER BY clause. If the ORDER BY clause is omitted, records are returned in arbitrary but consistent order.

```
<Page>         ::=  PAGE(<int>,<int>)
```

Expressions define the derived values that can be computed for result records:

```
<Expr>         ::=  <AggrExpr>
               ::=  <Expr>[+|-|*|/]<Expr>
               ::=  UnaryFunc(<Expr>)
               ::=  BinaryFunc(<Expr>, <Expr>)
               ::=  COALESCE(<Expr>, <Expr>,...)
<AggrExpr>     ::=  <SimpleExpr>
               ::=  <AggrFunc>(<SimpleExpr>) [<Where>]
<SimpleExpr>   ::=  <Key>
               ::=  <Literal>
               ::=  <SimpleExpr> [+|-|*|/] <SimpleExpr>
               ::=  <UnaryFunc>(<SimpleExpr>)
               ::=  <BinaryFunc>(<SimpleExpr>, <SimpleExpr>)
               ::=  <TimeDateFunc>(<SimpleExpr>,<DateTimeUnit>)
               ::=  <LookupExpr>
<AggrFunc>     ::=  ARB | AVG | COUNT | COUNTDISTINCT | MAX |
                    MEDIAN | MIN | STDDEV | SUM | VARIANCE
<UnaryFunc>    ::=  ABS | CEIL | COS | EXP | FLOOR | LN |
                    ROUND | SIGN | SIN | SQRT |
                    TAN | TO_DURATION | TRUNC
<BinaryFunc>   ::=  DIVIDE | LOG | MINUS | MOD | MULTIPLY |
                    PLUS | POWER | ROUND | TRUNC
```

```
<TimeDateFunc> ::=  EXTRACT | TRUNC
<DateTimeUnit> ::=  SECOND | MINUTE | HOUR | DAY_OF_MONTH |
                    DAY_OF_WEEK | DAY_OF_YEAR | DATE | WEEK |
                    MONTH | QUARTER | YEAR
```

Optionally, a WHERE clause may be specified after an aggregation function. As in SQL, the Analytics API WHERE clause expresses record filtering. But in addition to per-statement WHERE clauses, the Analytics API allows WHERE clauses to be specified at the expression level, allowing filtering of aggregate members for the computation of derived values on member subsets. SQL requires join operations to achieve a similar effect, with additional generality, but at the cost of efficiency and overall query execution complexity.

Lookup expressions provide the ability to refer to values in record sets that have been previously computed (possibly different from the current and FROM record set). Lookups can only refer to record sets that were grouped (non-empty GROUP BY clause) and the LookupList must match the GROUP BY fields of the lookup record set.

```
<LookupExpr>   ::=  <Key>[<LookupList>].<Key>
<LookupList>   ::=  <empty>
               ::=  <SimpleExpr> [,<LookupList>]
```

Note

For a specific example of a lookup expression in action, see the topic "Inter-statement references".

Filters provide basic comparison, range, membership and Boolean filtering capabilities, for use in WHERE and HAVING clauses.

```
<Filter>       ::=  DVAL(DvalID)
               ::=  <Key> <Compare> <Literal>
               ::=  <Key> IS [NOT] NULL
               ::=  <Filter> AND <Filter>
               ::=  <Filter> OR <Filter> | NOT <Filter>
               ::=  [<KeyList>] IN <Key>
<Compare>      ::=  = | <> | < | > | <= | >=
<KeyList>      ::=  <Key> [, <KeyList>]
<DvalID>       ::=  <int>
```

The IN filter can be used to filter data based on membership in a group or based on membership in another statement. It can only refer to previously computed record sets based on a non-empty GROUP BY. The number and type of keys in the KeyList must match the number and type of keys used in the statement referenced by the IN clause.

This query shows how the IN filter can be used to populate a pie chart showing sales divided into six segments: one segment for each of the five largest customers, and one segment showing the aggregate sales for all other customers. The first statement gathers the sales for the top five customers, and the second statement aggregates the sales for all customers not in the top five.

```
RETURN Top5 AS SELECT
SUM(Sale) AS Sales
GROUP BY Customer
ORDER BY Sales DESC
PAGE(0,5);
```

```
RETURN Others AS SELECT
SUM(Sale) AS Sales
WHERE NOT [Customer] IN Top5
GROUP
```

The WHERE IN clause does not respect multi-assign properties. In particular, the WHERE clause considers only a single value from a set of values in the multi-assign property. Therefore, if you want to filter on a property that is multi-assign, consider using the HAVING clause with the IN filter, instead of the WHERE IN clause. The following examples illustrates this case.

The following query uses the WHERE IN clause:

```
The original DEFINE Top100Terms AS SELECT
COUNT(1) AS Total
GROUP BY Terms
ORDER BY Total DESC PAGE(0,100);

RETURN DateTerms AS SELECT
COUNT(1) AS Total
WHERE [Terms] IN Top100Terms
GROUP BY Terms, Month
```

This query (above) considers only a single value from a set of values in the multi-assign property. Therefore, if you want to filter on a property that is multi-assign, consider using the following query, as a workaround (this workaround has a performance impact since the MDEX Engine will filters calculated results rather than performing filtering before computation):

```
DEFINE Top100Terms AS SELECT
COUNT(1) AS Total
GROUP BY Terms
ORDER BY Total DESC PAGE(0,100);

RETURN DateTerms AS SELECT
COUNT(1) AS Total
GROUP BY Terms, Month
HAVING [Terms] IN Top100Terms
```

Related links

- Inter-statement references (page 15)

Characters

All Unicode characters are accepted.

```
<Literal>      ::=  <StringLiteral> | <NumericLiteral>
```

String literals must be surrounded by single-quotes. Embedded single-quotes and backslashes must be escaped by backslashes. Numeric literals can be integers or floating point numbers. However, they do not support exponential notation, and they cannot have trailing f|F|d|D to indicate float or double.

Some example literals:

```
34   .34   'jim'   'àlêx\'s house'
```

Identifiers can either be quoted (using double-quote characters) or unquoted.

```
<Key>          ::=  <Identifier>
```

Unquoted identifiers can contain only letters, digits, and underscores, and cannot start with a digit. Quoted identifiers can contain any character, but double-quotes and backslashes must be escaped using a backslash.

To make some text be interpreted as an identifier instead of as a keyword, simply place it within quotation marks. For example if you have a property named WHERE or GROUP, you can reference it as "WHERE" or "GROUP". (Omitting the quotation marks would lead to a syntax error.)

Some example identifiers:

```
àlêx4
"4th street"
"some ,*#\" funny \\;% characters"
```

3 Charting API

This section describes the Endeca Analytics Charting API.

Related links

- About the Charting API (page 31)
- Grid class (page 31)

About the Charting API

The Endeca API extensions for Analytics provide several components that support rendering Analytics query results using the Grid class.

The Analytics API provides functions for computing a rich set of analytics, the results of which are returned as collections of Oracle Commerce Guide Search records. In most analytical applications, these result records are primarily intended for visual display in a chart, graph, or table. The Endeca API extensions for Analytics provide the following components that support rendering analytics query results:

Grid class A class to translate analytics records into a basic multi-dimensional array, included as part of the Charting API.

Further details about the methods associated with these classes can be found in the *Presentation API for Java (Javadoc)* and in the *Presentation and Logging API for .NET Reference (HTML Help)*.

Grid class

The Grid class provided by the Charting API presents a multi-dimensional array facade to a set of analytics result records. It can be used for application layer rendering (such as building tabular displays).

Typically, analytics statements return a list of records. Each result record contains property and/or dimension values representing the GROUP BY values for the bucket along with dynamically computed properties representing the results of Analytics expressions.

For example, an analytics query such as:

```
RETURN Results AS
SELECT COUNT(TransId) AS TransCount
GROUP BY Region
```

returns records of the form { Region, TransCount }:

```
{ Region="NorthEast", TransCount=20 }
{ Region="SouthEast", TransCount=35 }
{ Region="Central",   TransCount=25 }
...
```

Results such as these, which are grouped by a single dimension, are convenient for rendering a tabular view. But multidimensional groupings are less convenient. For example, consider a query such as:

```
RETURN Results AS
SELECT COUNT(TransId) AS TransCount
GROUP BY Region, Year
```

which returns records of the form { Region, Year, TransCount }, for example:

```
{ Region="NorthEast", Year="2009", TransCount=8  }
{ Region="NorthEast", Year="2010", TransCount=12 }
{ Region="SouthEast", Year="2010", TransCount=35 }
{ Region="Central",   Year="2009", TransCount=25 }
...
```

This data complicates table or chart rendering. It is not necessarily organized in an order convenient for the rendering code. Furthermore, the sparseness of the results (for example, because there were no 2009 sales in the SouthEast, no result record is returned for that grouping) must be handled.

The Grid class presents a multi-dimensional array facade to a set of analytics result records, providing an interface that is significantly more convenient for application layer rendering (such as building tabular displays).

The Grid constructor requires three parameters:

- Iterator—An iterator over ERec objects providing access to the records that are used to populate the Grid, typically obtained from a analytics result (such as the `AnalyticsStatementResult.getERecIter` method)
- String—Specifies which property in the input records should be used to populate the cell values in the grid, typically the SELECT AS key for the associated analytics query (such as TransCount) – in other words, the computed property that is being rendered
- List—A list of strings specifying the properties and/or dimensions in the input records that should be used as the grid axes, typically the GROUP BY keys for the associated analytics query (such as Region and Year)

For example, to create a Grid over the results of the above example, use the following code:

```
List axes = new List();
axes.add("Region");
axes.add("Year");
Grid grid = new Grid(analyticsStatementResult.getERecIter(),
```

```
  "TransCount",axes);
```

Convenience constructors are provided for one- and two-dimensional grids, so this example could also have been written:

```
Grid grid = new Grid(analyticsStatementResult.getERecIter(),
  "TransCount","Region","Year");
```

An initialized Grid provides a multidimensional array interface. It provides access to "label" values (that is, the set of values found in any of the axes specified in the constructor). For example:

```
List regions = grid.getLabels(0);
List years = grid.getLabels(1);
```

The returned Lists contain Label objects, each of which contains a property value (String) or a dimension value (DimVal). Given a list representing an array index (that is, a list of values, one from each axis), the Grid returns a Cell object representing the value at that location. For example:

```
List location = new List();
location.add(regions.get(0));
location.add(years.get(0));
Cell cell = grid.getValue(location);
```

Convenience accessors are provided for one- and two-dimensional grids, so this example could also have been written simply as:

```
Cell cell = grid.getValue(regions.get(0),years.get(0));
```

Cells contain the value for the grid at the specified location (a String), and also allow reverse lookups. Therefore, from a cell, one can get the list of labels for the location of the cell in its containing grid with the call `cell.getLabels()`.

Rendering an HTML table

One common use of the Grid class is for rendering HTML tables containing analytics results.

The following JSP code snippet provides an example of this usage for our example two-dimensional grid.

This example assumes that the Region and Year are navigation dimensions, not properties (hence the use of the getDimVal method on the labels in the axes).

```
<%
// Get X and Y labels
Grid grid = new Grid(iterator,"TransCount","Region","Year");
List regions = grid.getLabels(0);
List years = grid.getLabels(1);

// Display header row
%><tr><td></td><%
```

```
for (int i=0; i<regions.size(); i++) {
  Label region = (Label)regions.get(i);
  %><td><%= region.getDimVal().getName() %></td><%
}
%></tr><%

// Display data rows
for (int i=0; i<years.size(); i++) {
  Label year = (Label)years.get(i);
  %><tr><td><%= year.getDimVal().getName() %></td><%
  for (int j=0; j<regions.size(); j++) {
    Label region = (Label)regions.get(j);
    Cell cell = grid.getValue(region, year);
    %><td><%= cell.getValue() %></td><%
  }
  %></tr><%
}
%>
```

which, for our example result set above, would render in the form:

	Northeast	**Central**	**...**	**Southeast**
2009	8	25		20
2010	12	17		35