# Oracle® Endeca Server

Query Language Reference

Version 2.3.0 • June 2012 • Revision A

ORACLE®

# Copyright and disclaimer

# Table of Contents

# Preface

Oracle® Endeca Server is the core search-analytical database. It organizes complex and varied data from disparate source systems into a faceted data model that is extremely flexible and reduces the need for up-front data modeling. This highly-scalable server enables users to explore data in an unconstrained and impromptu manner and to rapidly address new questions that inevitably follow every new insight.

## About this guide

This guide describes how to write queries in the Endeca Query Language, or EQL.

## Who should use this guide

This guide is intended for data developers who need to create EQL queries.

## 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 Customer Support

Oracle Endeca Customer Support provides registered users with important information regarding Oracle Endeca software, implementation questions, product and solution help, as well as overall news and updates.

You can contact Oracle Endeca Customer Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1

# Introduction to the Endeca Query Language

This section introduces the Endeca Query Language (or EQL) and walks you through the query processing model.



## EQL overview

EQL is a SQL-like language designed specifically to query and manipulate data from the Oracle Endeca Server. It enables Endeca Server–based applications to examine aggregate information such as trends, statistics, analytical visualizations, comparisons, and more.

An EQL query contains one or more statements, each of which can group, join, and analyze records, either those stored in the server or those produced by other statements. Multiple statements within a single query can return results back to the application, allowing complex analyses to be done within a single query.

## Important concepts and terms

In order to work with EQL, you need to understand the following concepts.

- **Attribute:** An attribute is the basic unit of a record schema. Attributes describe records in the Endeca Server.
  - **Multi-assign attribute:** An attribute for which a record may have more than one value. For example, because a book may have more than one author, the Author attribute would be multi-assign.
  - **Managed attribute:** An attribute for which a hierarchy of attribute values is attached. Managed attributes are used to support hierarchical navigation.
  - **Standard attribute:** An attribute whose value is not included in an enumerated list or hierarchy.
- **Record:** The fundamental unit of data in the Endeca Server. Records are assigned attribute values. An assignment indicates that a record has a value for an attribute. A record typically has assignments from multiple attributes.

  **Note:** Records in the corpus can include multiple assignments to the same attribute. Records in EQL results cannot.

- **Corpus:** The full body of Endeca Server records. Endeca Server data is corpus–based rather than table–based. By default, the source of records for an EQL statement is the result of the containing search and navigation query. However, you can also include the `FROM` syntax in your statement to specify a different record source, either from the corpus or from a previously defined statement. Two names identify a corpus-based source:
  - `AllBaseRecords`: Every record that passed the security filter.
  - `NavStateRecords`: Every record that passed all previous filters.

  The omission of the `FROM` clause implies `FROM NavStateRecords`. This implicit `FROM` is equivalent to using a `WHERE` clause that expresses the filters currently applied.
- **Statement:** A unit of EQL that computes related or independent analytics results. In EQL, a statement starts with `DEFINE` or `RETURN` and ends with a semi-colon, and includes a mandatory `SELECT` clause and, optionally, some other clause(s).
- **Result:** Query results are a collection of statement results; statement results are a collection of records.
  - **Intermediate results:** Results from `RETURN` statements can also be used as intermediate results for further processing by other statements.
  - **Returned results:** Set of matching values returned by the query or statement.
- **Query:** A request sent to the Endeca Server. In general, a query consists of multiple statements.

# EQL and SQL: a comparison

EQL is, in many ways, similar to SQL, but has some marked differences as well.

This topic identifies EQL concepts that may be familiar to users familiar with SQL, as well as the unique features of EQL:

- **Tables with a single schema vs a corpus of records with more than one schema**. SQL is designed around tables of records — all records in a table have the same schema. EQL is designed around a single corpus of records with heterogeneous schemas.
- **EQL Query vs SQL Query**. An EQL statement requires a `DEFINE` or `RETURN` clause, which, like a SQL common table expression (or CTE), defines a temporary result set. The following differences apply, however:
  - EQL does not support a schema declaration.
  - In EQL, the scope of a CTE is the entire query, not just the immediately following statement.
  - In EQL, a `RETURN` is both a CTE and a normal statement (one that produces results).
  - EQL does not support recursion. That is, a statement cannot refer to itself using a `FROM` clause, either directly or indirectly.
  - EQL does not contain an update operation.
- **Clauses**. In EQL, `SELECT`, `FROM`, `WHERE`, `HAVING`, `GROUP BY`, and `ORDER BY` are all like SQL, with the following caveats:
  - In `SELECT` statements, `AS` aliasing is required in EQL; it is optional in SQL.
  - In EQL, `GROUP BY` implies `SELECT`. That is, grouping attributes are always included in statement results, whether or not they are explicitly selected.

  - Grouping by a multi-assign attribute can cause a single record to participate in multiple groups.
  - `GROUP BY` discards NULLs. (That is, input records that have no assignment for one or more grouping attributes are discarded.) This is different from SQL, which treats NULL like any other value.
  - `WHERE` can be applied to an aggregation expression.
  - In SQL, use of aggregation implies grouping. In EQL, grouping is always explicit.
- **Other language differences**.
  - `PAGE` works in the same way as many common vendor extensions to SQL.
  - In EQL, a `JOIN` expression's Boolean join condition must be contained within parentheses. This is not necessary in SQL.
  - EQL supports `SELECT` statements only. It does not support other DML statements, such as `INSERT` or `DELETE`, nor does it support DDL, DCL, or TCL statements.
  - EQL supports a different set of data types, expressions, and functions than described by the SQL standard.

# How queries are processed

This topic walks you through the steps involved in EQL query processing.

**Note:** This abstract processing model is provided for educational purposes and is not meant to reflect actual query evaluation.

Prior to processing each statement, EQL computes source records for that statement. When the records come from a single statement or the corpus, the source records are the result records of the statement or the appropriately filtered corpus records, respectively. When the records come from a `JOIN`, there is a source record for every pair of records from the left and right sides for which the join condition evaluates to true on that pair of records. Before processing, statements are re-ordered, if necessary, so that statements are processed before other statements that depend on them.

EQL then processes queries in the following order. Each step is performed within each statement in a query, and each statement is done in order:

1. It filters source records (both statement and per-aggregate) according to the `WHERE` clauses.
2. For each source record, it computes `SELECT` clauses that are used in the `GROUP BY` clause (as well as `GROUP BY`s not from `SELECT`s) and arguments to aggregations.
3. It maps source records to result records and computes aggregations.
4. It finishes computing `SELECT`s.
5. It filters result records according to the `HAVING` clause.
6. It orders result records.
7. It applies paging to the results.

# Chapter 2

# EQL Syntax and Semantics

This section describes EQL structure, syntax, and semantics.



## Query overview

An EQL query contains one or more semicolon-delimited statements.

Any number of statements from the query can return results, while others are defined only as generating intermediate results.

Each statement must contain at least two clauses: a `DEFINE` or a `RETURN` clause, and a `SELECT` clause. In addition, it may contain other, optional clauses.

Most clauses can contain expressions. Expressions are typically combinations of one or more functions, attributes, constants, or operators. Most expressions are simple combinations of functions and attributes. EQL provides functions for working with numeric, string, dateTime, duration, Boolean, and geocode attribute types.

Input records, output records, and records used in aggregation can be filtered in EQL. EQL supports filtering on arbitrary, Boolean expressions.

### About the examples in this section

Several of the examples in this section are based on sales data from a fictitious bicycle seller. The schema used matches the schema used in the Quick Start application, a reference implementation of Oracle Endeca Information Discovery. You can use these examples in the Quick Start application to begin experimenting with EQL.

### Syntax conventions used in this section

The syntax descriptions in this section use the following conventions:

| Convention | Meaning | Example |
|---|---|---|
| Square brackets [ ] | Optional | `FROM <statementKey> [alias]` |

| Convention | Meaning | Example |
|---|---|---|
| Asterisk * | May be repeated | [, JOIN statement [alias] ON <Boolean expression>]* |
| Ellipsis ... | Additional, unspecified content | DEFINE *<recordSetName>* AS ... |
| Angle brackets < > | Variable name | HAVING ***<Boolean expression>*** |

### Commenting in EQL

You can comment your EQL code using the following notation:

```
DEFINE Example AS SELECT /* This is a comment */
```

You can also comment out lines or sections as shown in the following example:

```
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
*/

...
```

**Note:** EQL comments cannot be nested.

## Statements and clauses

EQL queries consist of statements. Statements, in their turn, can contain several types of clauses.

- *DEFINE and RETURN clauses on page 6* specify whether to return the result.
- *SELECT clauses on page 6* specify how to compute attributes that will appear in statement results.
- *FROM clauses on page 7* (optional) specify the source.
- *JOIN clauses on page 8* (optional) allow source record joining.
- *WHERE clauses on page 11* (optional) filter source records.
- *GROUP/GROUP BY clauses on page 12* (optional) specify source record to result record mapping.
- *HAVING clauses on page 14* (optional) filter result records.
- *ORDER BY clauses on page 14* (optional) specify sort criteria.

- *PAGE clauses on page 15* (optional) specify the subset of records to be included in a statement result.

## DEFINE and RETURN clauses

All EQL statements begin with either DEFINE or RETURN. DEFINE is used to generate an intermediate result that will not be included in the query result. RETURN indicates that the statement result should be included in the query result.

### DEFINE

You can use multiple DEFINE clauses to make results available to other statements. Typically, DEFINE clauses are used to look up values, compare attribute values to each other, and normalize data.

Its syntax is as follows:

```
DEFINE <recordSetName> AS ...
```

In the following example, the RegionTotals record set is used in a subsequent calculation:

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

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

### RETURN

RETURN provides the key for accessing EQL results from the Endeca Server query result. This is important when more than one statement is submitted with the query.

Its syntax is as follows:

```
RETURN <recordSetName> AS ...
```

The following statement returns for each size the number of different values for the Color attribute:

```
RETURN result AS
SELECT COUNTDISTINCT(Color) AS Total
GROUP BY Size
```

## SELECT clauses

The SELECT clause defines the list of attributes on the records produced by the statement.

Its syntax is as follows:

```
SELECT <expression> AS <attribute key>[, <expression> AS <key>]*
```

For example:

```
SELECT Sum(Amount) AS TotalSales
```

The attribute definitions can refer to previously defined attributes, as shown in the following example:

```
SELECT Sum(Amount) AS TotalSales, TotalSales / 4 AS QuarterAvg
```

**Note:** If an attribute defined in a `SELECT` clause is used in the statement's `GROUP` clause, then the expression can only refer to source attributes and other attributes used in the `GROUP` clause. It must not contain aggregations.

## FROM clauses

You can include a `FROM` clause in your statement to specify a different record source than the result of the containing search and navigation query.

Its syntax is as follows:

```
FROM <statementKey> [alias]
```

By default, the source of records for an EQL statement is the result of the containing search and navigation query. However, you can also include the `FROM` syntax in your statement to specify a different record source, either from the corpus or from a previously defined statement, whether that statement is a `DEFINE` or a `RETURN`.

Two names identify a corpus-based source:

- `AllBaseRecords`: Every record that passed the security filter.
- `NavStateRecords`: Every record that passed all previous filters.

**Note:** If you want to submit your query against `NavStateRecords`, you do not need to include the `FROM` syntax in your statement. The absence of `FROM` implies `NavStateRecords`.

You can also use the result of a different statement as your record source. In the following example, a statement computes the total number of sales transactions for each quarter and sales representative. To then compute the average number of transactions per sales rep, a subsequent statement groups those results by quarter.

```
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
```

The `RepQuarters` statement generates a list of records. Each record contains the attributes `{ SalesRep, Quarter, NumTrans }`. For example:

```
{ J. Smith, 11Q1, 10 }
{ J. Smith, 11Q2, 3 }
{ F. Jackson, 10Q4, 10 }
...
```

The `Quarters` statement then uses the results of the `RepQuarters` statement to generate a list with the attributes `{ Quarter, AvgTransPerRep }`. For example:

```
{ 10Q4, 10 }
{ 11Q1, 4.5 }
{ 11Q2, 6 }
...
```

# JOIN clauses

JOIN clauses allow records from multiple statements to be combined.

JOIN clauses, which conform to a subset of the SQL standard, do a join with the specified join condition. The join condition may be an arbitrary Boolean expression referring to the attributes in the FROM statement. The expression must be enclosed in parentheses.

The JOIN clause always modifies a FROM clause. Two named sources can be indicated in the FROM clause. Fields must be dot-qualified to indicate which source they come from, except in queries from a single table.

Self-join is supported. Statement aliasing is required for self-join.

Both input tables must result from DEFINE or RETURN statements (that is, from intermediate results). AllBaseRecords and NavStateRecords cannot be joined.

Any number of joins can be performed in a single statement.

The syntax of JOIN is as follows:

```
FROM <Statement> [alias]
[LEFT,RIGHT,FULL] JOIN <Statement2> [alias]
ON (Boolean expression) [, JOIN <StatementN> [alias] ON (Boolean expression)]*
```

If there is more than one JOIN, each statement is joined with a FROM statement.

## Types of joins

EQL supports the following types of joins:

- **INNER JOIN:** INNER JOIN joins records on the left and right sides, then filters the result records by the join condition. That means that only rows for which the join condition is TRUE are included. If you do not specify the join type, JOIN defaults to INNER JOIN.
- **LEFT JOIN**, **RIGHT JOIN**, and **FULL JOIN:** LEFT JOIN, RIGHT JOIN, and FULL JOIN (collectively called *outer joins*) extend the result of an INNER JOIN with records from a side for which no record on the other side matched the join condition. When such an additional record is included from one side, the record in the join result contains NULLs for all attributes from the other side. LEFT JOIN includes all such rows from the left side, RIGHT JOIN includes all such rows from the right side, and FULL JOIN includes all such rows from either side.
- **CROSS JOIN:** The result of CROSS JOIN is the Cartesian product of the left and right sides. Each result record has the assignments from both of the corresponding records from the two sides.

  **Important:** CROSS JOIN should be used with caution, because it can generate very large numbers of records. For example, a CROSS JOIN of a result with 100 records and a result with 200 records would contain 20,000 records.

## JOIN examples

The following INNER JOIN example finds employees whose sales in a particular subcategory account for more than 10% of that subcategory's total:

```
DEFINE EmployeeTotals AS
SELECT
   DimEmployee_FullName AS Name,
   SUM(FactSales_SalesAmount) AS Total
GROUP BY DimEmployee_EmployeeKey, ProductSubcategoryName;

DEFINE SubcategoryTotals AS
```

```
SELECT
   SUM(FactSales_SalesAmount) AS Total
GROUP BY ProductSubcategoryName;

RETURN Stars AS
SELECT
   EmployeeTotals.Name AS Name,
   EmployeeTotals.ProductSubcategoryName AS Subcategory,
   100 * EmployeeTotals.Total / SubcategoryTotals.Total AS Pct
FROM EmployeeTotals
   JOIN SubcategoryTotals
   ON (EmployeeTotals.ProductSubcategoryName = SubcategoryTotals.ProductSubcategoryName)
HAVING Pct > 10
```

The following self-join using `INNER JOIN` computes cumulative daily sales totals per employee:

```
DEFINE Days AS
SELECT
   FactSales_OrderDateKey AS DateKey,
   DimEmployee_EmployeeKey AS EmployeeKey,
   DimEmployee_FullName AS EmployeeName,
   SUM(FactSales_SalesAmount) AS DailyTotal
GROUP BY DateKey, EmployeeKey;

RETURN CumulativeDays AS
SELECT
   SUM(PreviousDays.DailyTotal) AS CumulativeTotal,
   Day.DateKey AS DateKey,
   Day.EmployeeKey AS EmployeeKey,
   Day.EmployeeName AS EmployeeName
FROM Days Day
   JOIN Days PreviousDays
   ON (PreviousDays.DateKey <= Day.DateKey)
GROUP BY DateKey, EmployeeKey
```

The following `LEFT JOIN` example computes the top 5 subcategories along with an Other bucket, for use in a pie chart:

```
DEFINE Totals AS
SELECT
   SUM(FactSales_SalesAmount) AS Total
GROUP BY ProductSubcategoryName;

DEFINE Top5 AS
SELECT
   Total AS Total
FROM Totals
GROUP BY ProductSubcategoryName
ORDER BY Total DESC PAGE(0,5);
RETURN Chart AS
SELECT
   COALESCE(Top5.ProductSubcategoryName, 'Other') AS Subcategory,
   SUM(Totals.Total) AS Total
FROM Totals
   LEFT JOIN Top5
   ON (Totals.ProductSubcategoryName = Top5.ProductSubcategoryName)
GROUP BY Subcategory
```

The following `LEFT JOIN` computes metrics for each product in a particular region, ensuring all products appear in the list even if they have never been sold in that region:

```
DEFINE Product AS
SELECT
   ProductAlternateKey AS Key,
   ProductName AS Name GROUP BY Key;

DEFINE RegionTrans AS
```

```
SELECT
   ProductAlternateKey AS ProductKey,
   FactSales_SalesAmount AS Amount
WHERE DimSalesTerritory_SalesTerritoryRegion='United Kingdom';


RETURN Results AS
SELECT
   Product.Key AS ProductKey,
   Product.Name AS ProductName,
   COALESCE(SUM(RegionTrans.Amount), 0) AS SalesTotal,
   COUNT(RegionTrans.Amount) AS TransactionCount
FROM Product
   LEFT JOIN RegionTrans
   ON (Product.Key = RegionTrans.ProductKey)
GROUP BY ProductKey
```

The following `FULL JOIN` computes the top 10 employees' sales totals for the top 10 products, ensuring that each employee and each product appears in the result:

```
DEFINE TopEmployees AS
SELECT
   DimEmployee_EmployeeKey AS Key,
   DimEmployee_FullName AS Name,
   SUM(FactSales_SalesAmount) AS SalesTotal
GROUP BY Key
ORDER BY SalesTotal DESC
PAGE (0,10);


DEFINE TopProducts AS
SELECT
   ProductAlternateKey AS Key,
   ProductName AS Name,
   SUM(FactSales_SalesAmount) AS SalesTotal
GROUP BY Key
ORDER BY SalesTotal DESC
PAGE (0,10);

DEFINE EmployeeProductTotals AS
SELECT
   DimEmployee_EmployeeKey AS EmployeeKey,
   ProductAlternateKey AS ProductKey,
   SUM(FactSales_SalesAmount) AS SalesTotal
GROUP BY EmployeeKey, ProductKey
HAVING [EmployeeKey] IN TopEmployees AND [ProductKey] IN TopProducts;


RETURN Results AS
SELECT
   TopEmployees.Key AS EmployeeKey,
   TopEmployees.Name AS EmployeeName,
   TopEmployees.SalesTotal AS EmployeeTotal,
   TopProducts.Key AS ProductKey,
   TopProducts.Name AS ProductName,
   TopProducts.SalesTotal AS ProductTotal,
   EmployeeProductTotals.SalesTotal AS EmployeeProductTotal
FROM EmployeeProductTotals
   FULL JOIN TopEmployees
   ON (EmployeeProductTotals.EmployeeKey = TopEmployees.Key)
   FULL JOIN TopProducts
   ON (EmployeeProductTotals.ProductKey = TopProducts.Key)
```

The following `CROSS JOIN` example finds the percentage of total sales each product subcategory represents:

```
DEFINE GlobalTotal AS
SELECT
```

```
   SUM(FactSales_SalesAmount) AS GlobalTotal
GROUP;

DEFINE SubcategoryTotals AS
SELECT
   SUM(FactSales_SalesAmount) AS SubcategoryTotal
GROUP BY ProductSubcategoryName;


RETURN SubcategoryContributions AS
SELECT
   SubcategoryTotals.ProductSubcategoryName AS Subcategory,
   SubcategoryTotals.SubcategoryTotal / GlobalTotal.GlobalTotal AS Contribution
FROM SubcategoryTotals
   CROSS JOIN GlobalTotal
```

**Important:** Joins can cause the Endeca Server to grow beyond available RAM. Going beyond the scale capabilities will cause very, very large materializations, intense memory pressure, and can result in an unresponsive Endeca Server.

## WHERE clauses

EQL provides two filtering options: `WHERE` and `HAVING`. The `WHERE` clause is used to filter input records for an expression.

Its syntax is as follows:

```
WHERE <Boolean expression>
```

You can use the `WHERE` clause with any Boolean expression, such as:

- Numeric and string value comparison: `{= , <>, <, <=. >, >=}`
- Null value evaluation: `<attribute> IS {NULL, NOT NULL}`
- Grouping keys of the source statement: `<attribute list> IN <source statement>`. The number and type of these keys must match the number and type of keys used in the statement referenced by the `IN` clause. For more information, see *IN on page 40*.

If an aggregation function is used with a `WHERE` clause, then the Boolean expression must be enclosed within parentheses. The aggregation functions are listed in the topic *Aggregation functions on page 26*.

In this example, the amounts are only calculated for sales in the West region. Then, within those results, only sales representatives who generated at least $10,000 are returned:

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

In the next example, a single statement contains two expressions. The first expression computes the total for all of the records and the second expression computes the total for one specific sales representative:

```
RETURN QuarterTotals AS SELECT
   SUM(Amount) As SalesTotal,
   SUM(Amount) WHERE (SalesRep = 'Juan Smith') AS JuanTotal
GROUP BY Quarter
```

This would return both the total overall sales and the total sales for Juan Smith for each quarter. Note that the Boolean expression in the `WHERE` clause is in parentheses because it is used with an aggregation function (`SUM` in this case).

## GROUP/GROUP BY clauses

The GROUP/GROUP BY clause specifies how to map source records to result records to group statement output.

There are three ways to use this clause in a query:

- Omitting the GROUP clause maps each source record to its own result record.
- GROUP maps all source records to a single result record.
- GROUP BY <attribute list> maps source records to result records by the combination of values in the listed attributes.

### Specifying only GROUP

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

For example, the following statement uses the SUM statement to return a single sum across a set of records:

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

This statement returns one record for NumReviews. The value is the sum of the values for the attribute number_of_reviews.

Grouping is allowed on source and locally defined attributes.

**Note:** If you group by a locally defined attribute, that attribute cannot refer to non-grouping attributes and cannot contain any aggregates.

All grouping attributes are part of the result records. A NULL value in any grouping attribute causes the source record to map to no result records. This is different from SQL, which treats NULL like any other value. For information about user-defined NULL-value handling in EQL, see *COALESCE on page 38*.

### Specifying GROUP BY

You can use GROUP BY to aggregate results into buckets with common values for the grouping keys.

For example, suppose we have sales transaction data with records consisting of the following attributes:

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

For example:

```
{ TransId = 1, ProductType = "Widget", Amount = 100.00,
  Year = 2011, Quarter = "11Q1", Region  = "East",
  SalesRep = "J. Smith", Customer = "Customer1" }
```

If an EQL statement uses Region and Year as GROUP BY attributes, the statement results contain an aggregated record for each valid, non-empty combination of Region and Year. In EQL, this example is expressed as:

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

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

```
{ "East", "2010" }
{ "West", "2011" }
```

```
{ "East", "2011" }
```

### Using a GROUP BY that is an output of a SELECT expression

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

For example, the following syntax is 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 (`SUM`):

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

### Specifying the hierarchy level for a managed attribute

You can group by a specified depth of each managed attribute.

If you group results by a managed attribute, you can specify a hierarchy depth at which to group, using the syntax:

```
GROUP BY ManagedAttr:<level>
```

For example, the Region attribute contains the hierarchy Country, State, and City. To group the results at the State level (one level below the root of the managed attribute hierarchy), you would use the following syntax:

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

**Note:** This is equivalent to `ANCESTOR(ManagedAttr, level)`, but `GROUP BY` statements need to use the syntax `managedAttr:level`, because you cannot group by an expression.

### Grouping by a multi-assign attribute

If you group by a multi-assign attribute, each source record will map to multiple corresponding output records. For example, the record `[A:1, A:2, B:3, B:4, B:5]` will map to:

- Two output records if you group by A
- Three output records if you group by B
- Six output records if you group by both A and B
- Six output records for `SELECT A + B AS C GROUP BY C`, because all six possible values of A + B will be computed prior to grouping.

This can only occur with a corpus source, because result records are always single assign.

In this example, `UserTag` is multi-assign:

```
RETURN "Example" AS SELECT
   AVG("Gross") AS "AvgGross",
   SUM("Gross") AS "TotalGross",
GROUP BY UserTag
```

To define the set of resulting buckets, a statement must specify a set of `GROUP BY` attributes. The cross product of all values in these grouping attributes defines the set of candidate buckets.

The results are automatically pruned to include only non-empty buckets.

If an attribute reference appears in a statement with a GROUP clause in the definition of an attribute not in the GROUP clause, the attribute will have an implicit ARB aggregate applied.

## HAVING clauses

EQL provides two filtering options: WHERE and HAVING. The HAVING clause is used to filter output records.

Its syntax is as follows:

```
HAVING <Boolean expression>
```

You can use the HAVING clause with any Boolean expression, such as:

- Numeric and string value comparison: {= , <>, <, <=. >, >=}
- Null value evaluation: <attribute> IS {NULL, NOT NULL}
- Grouping keys of the source statement: <attribute list> IN <source statement>

In the following example, the results include only sales representatives who generated at least $10,000:

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

## ORDER BY clauses

The ORDER BY clause is used to control the order of result records.

The ORDER BY syntax is as follows:

```
ORDER BY <AttrName> [ASC/DESC] [,<AttrName> [ASC/DESC]]*
```

You can sort result records by any of their attribute values, and can specify whether to sort in ascending (ASC) or descending (DESC) order. You can use any combination of values and sort orders. The absence of a direction, as with Attr3 in the example above, implies ASC.

When an ORDER BY clause is used, NULL values will always sort after non-NULL values for a given attribute, and NaN (not-a-number) values will always sort after values other than NaN and NULL, regardless of the direction of the sort. Tied ranges (or all records in the absence of an ORDER BY clause) are ordered in an arbitrary but stable way: the same query will always return its results in the same order, as long as it is querying against the same version of the data. Data updates add or remove records from the order, but will not change the order of unmodified records.

In this example, the amount is calculated for each sales representative. The resulting records are sorted by total amount in descending order:

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

### Geocode sorting

Data of type `geocode` is sorted by latitude and then by longitude. To establish a more meaningful sort order when using `geocode` data, compute the distance from some point and then sort by the distance.

### String sorting

String values are sorted in Unicode byte order.

### Stability of ORDER BY

EQL guarantees that the results of a statement are stable across queries. This means that:

- If no updates are performed, then the same statement will return results in the same order on repeated queries, even if no `ORDER BY` clause is specified, or there are ties in the order specified in the `ORDER BY` clause.
- If updates are performed, then only changes that explicitly impact the order will impact the order; the order will not be otherwise affected. Changes that impact the order are things like deleting or inserting records that contribute to the result on or prior to the returned page, or modifying a value that is used for grouping or ordering.

For example, on a statement with no `ORDER BY` clause, queries that use `PAGE(0, 10)`, then `PAGE(10, 10)`, then `PAGE(20, 10)` will, with no updates, return successive groups of 10 records from the same arbitrary but stable result.

For an example with updates, on a statement with `ORDER BY Num PAGE(3, 4)`, an initial query returns records {5, 6, 7, 8}. An update then inserts a record with 4 (before the specified page), deletes the record with 6 (on the specified page), and inserts a record with 9 (after the specified page). The results of the same query, after the update, would be {4, 5, 7, 8}. This is because:

- The insertion of 4 shifts all subsequent results down by one. Offsetting by 3 records includes the new record.
- The removal of 6 shifts all subsequent results up by one.
- The insertion of 9 does not impact any of the records prior to or included in this result.

## PAGE clauses

The `PAGE` clause specifies a subset of records to return.

By default, a statement returns all of the result records. In some cases, however, it is useful to request only a subset of the results. In these cases, you can use the `PAGE (<offset>, <count>)` clause to specify how many result records to return.

The `<offset>` argument is an integer that determines the number of records to skip. An offset of 0 will return the first result record; an offset of 8 will return the ninth. The `<count>` argument is an integer that determines the number of records to return.

The following example groups the `NavStateRecords` by the SalesRep attribute, and returns result records 11-20:

```
DEFINE Reps AS
GROUP BY SalesRep
Page (10,10)
```

`PAGE` applies to intermediate results; a statement `FROM` a statement with `PAGE(0, 10)` will have at most 10 source records.

### Top-k

You can use the `PAGE` clause in conjunction with the `ORDER BY` clause in order to create Top-K queries. The following example returns the top 10 sales representatives by total sales:

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

### Percentile

The `PAGE` clause supports a `PERCENT` modifier. When `PERCENT` is specified, fractional offset and size are allowed, as in the example `PAGE(33.3, 0.5) PERCENT`. This specified the portion of the data set to skip and the portion to return.

The number of records skipped equals `round(offset * COUNT / 100)`.

The number of records returned equals `round((offset + size) * COUNT / 100) - round(offset * COUNT / 100)`.

```
DEFINE  "ModelYear" AS
SELECT  SUM(Cost) AS Cost
GROUP BY Model, Year
ORDER BY Cost DESC
PAGE(0, 10) PERCENT
```

The `PERCENT` keyword will not repeat records at non-overlapping offsets, but the number of results for a given page size may not be uniform across the same query.

For example, if `COUNT` = 6:

| PAGE clause | Resulting behavior is the same as |
|---|---|
| PAGE (0, 25) PERCENT | PAGE (0, 2) |
| PAGE (25, 25) PERCENT | PAGE (2, 1) |
| PAGE (50, 25) PERCENT | PAGE (3, 2) |
| PAGE (75, 25) PERCENT | PAGE (5, 1) |

# Grouping and aggregation

In EQL, aggregation operations bucket a set of records into a resulting set of aggregated records.

## Nested aggregation

You can perform multi-level aggregation in EQL.

This example computes 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 representatives to get per-representative transaction counts. Then these representative 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
```

## Using the COUNT and COUNTDISTINCT functions

The `COUNT` function returns the number of records that have a value for an attribute. `COUNTDISTINCT` counts the number of distinct values for an attribute.

### Using COUNT to count the number of records with values of attributes

The `COUNT` function counts the number of records that have values in a field for each `GROUP BY` result.

For example, the following records include Size and Color attributes:

```
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 following statement returns the number of records for each size that have a value for the Color attribute:

```
RETURN result AS SELECT COUNT(Color) as Total GROUP BY Size
```

The statement result is:

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

Because all of the records have the same value for Size, there is only one group, and thus only one record. For this group, the value of Total is 3, because only three of the records have Color assignments.

### Using COUNTDISTINCT to get the number of distinct values for an attribute

The `COUNTDISTINCT` function returns the number of unique values in a field for each `GROUP BY` result.

`COUNTDISTINCT` can only be used for single-assign attributes, and not for multi-assigned attributes. Using a multi-assign attribute generates misleading results.

For example, for the following records:

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

The following statement returns for each size the number of different values for the Color attribute:

```
RETURN result AS
SELECT COUNTDISTINCT (Color) as Total
GROUP BY Size
```

The statement result is:

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

Because all of the records have the same value for Size, there is only one group, and thus only one record. For this group, the value of Total is 2 because there are two unique values for the Color attribute: red and blue.

## Per-aggregation filters

Each aggregation can have its own filtering `WHERE` clause. Aggregation function filters filter the inputs to an aggregation expression. They are useful for working with sparse or heterogeneous data. Only records that satisfy the filter contribute to the calculation of the aggregation function.

The syntax is as follows:

```
AggregateFunction(Expression) WHERE (Filter)
```

For example:

```
RETURN NetSales AS SELECT
  SUM(Amount) WHERE (Type=‘Sale’)
    AS SalesTotal,
  SUM(Amount) WHERE (Type=‘Return’)
    AS ReturnTotal,
  SalesTotal – ReturnTotal AS Total
GROUP BY Year, Month, Category
```

This is the same as:

```
SUM(CASE WHEN Type='Sale' THEN Amount END) AS SalesTotal,
SUM(CASE WHEN type='Return' THEN Amount END) AS ReturnTotal
...
```

**Note:** These `WHERE` clauses also operate on records, not assignments, just like the statement-level `WHERE` clause. A source record will contribute to an aggregation if it passes the statement-level `WHERE` clause and the aggregation's `WHERE` clause.

## Handling of records with multiple values for an attribute

In the case of corpus records (but not result records) an attribute may allow a record to have multiple values.

To show how EQL handles these types of records, for a record tagged with both `Blue` and `Green`:

- `WHERE Color = Blue` matches the record (`Blue = Blue`)
- `WHERE Color <> Blue` matches the record (`Green <> Blue`)

- `WHERE NOT(Color = Blue)` does not match the record (`NOT(true)`)
- `WHERE NOT(Color <> Blue)` does not match the record (`NOT(true)`)

# Expressions

Expressions are typically combinations of one or more functions, attributes, constants, or operators. Most expressions are simple combinations of functions and attributes.

## Supported data types

This topic describes the format of data types supported by EQL.

| Data type | Description |
| --- | --- |
| string | Represents character strings. |
| int | Represents an integer.<br>**Note:** EQL supports only 64-bit integers. |
| double | Represents a floating point number. |
| boolean | Represents a Boolean value (`TRUE` or `FALSE`) |
| time | Represents the time of day to a resolution of milliseconds. |
| dateTime | Represents a date and time to a resolution of milliseconds. |
| duration | Represents a length of time with a resolution of milliseconds. |
| geocode | Represents a latitude and longitude pair. |

## Operator precedence rules in EQL

EQL enforces the following precedence rules for operators.

The rules are listed in descending order.

- Parentheses (as well as brackets in `LOOKUP` and `IN` expressions)

  **Note:** You can freely add parentheses any time you want to impose an alternative precedence or to make precedence clearer.

- * /
- + -
- = <> < > <= >=

- `IS` (`IS NULL`, `IS NOT NULL`)
- `BETWEEN`
- `NOT`
- `AND`
- `OR`

**Note:** All binary operators are left-associative, as are all of the `JOIN` operators.

# Literals

This section discusses how literals are used in EQL.

## Handling of characters in EQL

EQL accepts all Unicode characters.

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

| | |
|---|---|
| **String literals** | String literals must be surrounded by single quotation marks.<br>Embedded single quotes and backslashes must be escaped by backslashes. Examples:<br>`'jim'`<br>`'àlêx\'s house'` |
| **Numeric literals** | Numeric literals can be integers or floating point numbers.<br>Numeric literals cannot be surrounded by single quotation marks.<br>Numeric literals do not support exponential notation, and they cannot have trailing f\|F\|d\|D to indicate float or double.<br>`34`<br>`.34` |
| **Boolean literal** | `TRUE/FALSE`<br>Boolean literals cannot be surrounded by single quotation marks. |
| **Literals of structured types (such as Date, Time, or Geocode)** | Literals of structured types must use appropriate conversions, as shown in the following example:<br>`RETURN Result AS`<br>`SELECT  TO_GEOCODE(45.0, 37.0) AS Geocode,`<br>`        TO_DATETIME('2011-11-21T08:22:00Z') AS Timestamp` |

| | |
|---|---|
| **Identifiers** | Identifiers must be NCNames. The NCName format is defined in the W3C document Namespaces in XML 1.0 (Second Edition), located at this URL: *http://www.w3.org/TR/REC-xml-names/*.<br>An identifier must be enclosed in double quotation marks if:<br>• The identifier contains characters other than letters, digits, and underscores.<br>• The identifier starts with a digit.<br>• The identifier uses the same name as an EQL keyword. For example, if an attribute is named `WHERE` or `GROUP`.<br>If an identifier is in quotation marks, then you must use a backslash to escape double quotation marks and backslashes.<br>Examples:<br>`"Count" "Sales.Amount"` |

## Handling of upper- and lower-case in EQL

This topic discusses character case handling in EQL.

The following are case sensitive:

- Identifiers
- Literals
- Standard attribute references
- Managed attribute references

The following are case insensitive:

- Clauses
- Reserved words
- Keywords

## Handling NULL attribute values

If an attribute value is missing for a record, then the attribute is referred to as being NULL. For example, if a record does not contain an assignment for a Price attribute, EQL defines the Price value as NULL.

The following table outlines how EQL handles NULL values for each type of operation:

| Type of operation | How EQL handles NULL values |
|---|---|
| Arithmetic operations and non-aggregating functions | The value of any operation on a NULL value is also defined as NULL.<br>For example, if a record has a value of 4 for Quantity and a NULL value for Price, then the value of `Quantity + Price` is considered to be NULL. |

| Type of operation | How EQL handles NULL values |
| --- | --- |
| Aggregating functions | EQL ignores records with NULL values.<br>For example, if there are 10 records, and 2 of them have a NULL value for a Price attribute, all aggregating operations ignore the 2 records, and instead compute their value using only the other 8 records.<br>If all 10 records have a NULL Price, then most aggregations such as `SUM(Price)` also result in NULL values.<br>The exceptions are `COUNT` and `COUNTDISTINCT`, which return zero if all the records have a NULL value. (That is, the output of `COUNT` or `COUNTDISTINCT` is never NULL.) |
| Grouping expressions | EQL ignores any record that has a NULL value in any of the group keys, and does not consider the record to be present in any group. |
| Filters | When doing a comparison against a specific value, the NULL value will not match the specified filter, except for the `IS NULL` filter.<br>For example, if record A has price 5, and record B has no price value, then:<br>• `WHERE price = 5` matches A<br>• `WHERE NOT(price = 5)` matches B<br>• `WHERE price <> 5` matches neither A nor B<br>• `WHERE NOT(price <> 5)` matches both A and B<br>• `WHERE price = 99` matches neither A nor B<br>• `WHERE NOT(price = 99)` matches both A and B<br>• `WHERE price <> 99` matches A<br>• `WHERE NOT(price <> 99)` matches B |
| Sorting | For any sort order specified, EQL returns:<br>1. Normal results<br>2. Records for a NaN value<br>3. Records with a NULL value |

**Note:** There is no NULL keyword or literal. To create a NULL, use `CASE`, as in this example: `CASE WHEN False THEN 1 END`.

## Type promotion in EQL

In general, EQL performs type promotion when there is no risk of loss of information.

For example, in the expression `1 + 3.5`, 1 is an integer and 3.5 is a double. The integer value is promoted to a double, and the overall result is 4.5.

Some functions, such as `LN()`, take double arguments, and automatically promote integer arguments to doubles. In most other cases, automatic type promotion is not performed, and an explicit conversion is required. For example, if Quantity is an integer and SingleOrder is a Boolean, then an expression such as the following is not allowed:

```
COALESCE(Quantity, SingleOrder)
```

An explicit conversion from Boolean to integer like the following is required:

```
COALESCE(Quantity, TO_INTEGER(SingleOrder))
```

## Handling of NaN, inf and -inf results

Operations in EQL adhere to the conventions for Not a Number (`NaN`), `inf` and `-inf` defined by the IEEE 7540 2008 standard for handling floating point numbers.

In cases when it has to perform operations involving floating point numbers, or operations involving division by zero or NULL values, EQL expressions can return `NaN`, `inf`, and `-inf` results.

For example, `NaN`, `inf` and `-inf` values could arise in your EQL calculations when:

- A zero divided by zero results in `NaN`
- A positive number divided by zero results in `inf`
- A negative number divided by zero results in `-inf`

For most operations, EQL treats `NaN`, `inf` or `-inf` values the same way as any other value.

However, you may find it useful to know how EQL defines the following special values:

| Type of operation | How EQL handles `NaN`, `inf`, and `-inf` |
|---|---|
| Arithmetic operations | Arithmetic operations with `NaN` values result in `NaN` values. |
| Filters | `NaN` values do not pass filters (except for `!=`).<br>Any other comparison involving a `NaN` value is false. |
| Sorting | `NaN` is treated as "less than" `-inf` (`NaN` < `-inf`).<br>For any sort order specified, EQL returns:<br>1. Normal records<br>2. Records with a `NaN` value<br>3. Records with a NULL value |

# Functions and operators

EQL contains a number of built-in functions that process data. It also supports arithmetic operators.

## Numeric functions

EQL supports the following numeric functions.

| Function | Description and Example |
|---|---|
| addition | The addition operator `(+)`.<br>`SELECT NortheastSales + SoutheastSales AS EastTotalSales` |
| subtraction | The subtraction operator `(-)`.<br>`SELECT SalesRevenue - TotalCosts AS Profit` |
| multiplication | The multiplication operator `(*)`.<br>`SELECT Price * 0.7 AS SalePrice` |
| division | The division operator `(/)`.<br>`SELECT YearTotal / 4 AS QuarterAvg` |
| ABS | Returns the absolute value of n.<br>If n is `0` or a positive integer, returns n.<br>Otherwise, n is multiplied by -1.<br>`SELECT ABS(-1) AS one`<br>**RESULT:** `one` = `1` |
| CEIL | Returns the smallest integer value not less than n.<br>`SELECT CEIL(123.45) AS x, CEIL(32) AS y, CEIL(-123.45) AS z`<br>**RESULT:** `x` = `124`, `y` = `32`, `z` = `123` |
| EXP | Exponentiation, where the base is e.<br>Returns the value of e (the base of natural logarithms) raised to the power n.<br>`SELECT EXP(1.0) AS baseE`<br>**RESULT:** `baseE` = `e^1.0` = `2.71828182845905` |
| FLOOR | Returns the largest integer value not greater than n.<br>`SELECT FLOOR(123.45 AS x, FLOOR(32) AS y, FLOOR(-123.45) AS z`<br>**RESULT:** `x` = `123`, `y` = `32`, `z` = `124` |

| Function | Description and Example |
|---|---|
| LN | Natural logarithm. Computes the logarithm of its single argument, the base of which is `e`.<br>`SELECT LN(1.0) AS baseE`<br>**RESULT:** `baseE` = `e^1.0` = `0` |
| LOG | Logarithm. `log(n, m)` takes two arguments, where `n` is the base, and `m` is the value you are taking the logarithm of.<br>`Log(10,1000) = 3` |
| MOD | Modulo. Returns the remainder of `n` divided by `m`.<br>`Mod(10,3) = 1`<br>EQL uses the `fmod` floating point remainder, as defined in the C/POSIX standard. |
| ROUND | Returns a number rounded to the specified decimal place.<br>The unary version drops the decimal (non-integral) portion of the input.<br>The binary version allows you to set the number of spaces at which the number is rounded:<br>• 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`<br>• 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.<br>`SELECT SIGN(-12) AS x, SIGN(0) AS y, SIGN(12) AS z`<br>**RESULT:** `x` = `-1`, `y` = `0`, `z` = `1` |
| SQRT | Returns the nonnegative square root of `n`.<br>`SELECT SQRT(9) AS x`<br>**RESULT:** `x` = `3` |

| Function | Description and Example |
| --- | --- |
| TRUNC | Returns the number `n`, truncated to `m` decimal places.<br>If `m` is 0, the result has no decimal point or fractional part.<br>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.<br>`SELECT TRUNC(3.14159265, 3)as x`<br>**RESULT:** `x = 3.141` |
| SIN | The sine of `n`, where the angle of `n` is in radians.<br>`SIN(Pi/6) = 5` |
| COS | The cosine of `n`, where the angle of `n` is in radians.<br>`COS(Pi/3) = .5` |
| TAN | The tangent of `n`, where the angle of `n` is in radians.<br>`TAN(Pi/4) = 1` |
| POWER | Returns the value of `n` raised to the power of `m`.<br>`Power(2,8) = 256` |
| TO_DURATION | Casts an integer into a number of milliseconds so that it can be used as a duration. |
| TO_DOUBLE | Casts an integer as a double. |
| TO_INTEGER(Boolean) | Casts `TRUE/FALSE` to `1/0`. |

## Aggregation functions

EQL supports the following aggregation functions.

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

| Function | Description |
|---|---|
| MIN | Finds the minimum value for a field. |
| MEDIAN | Finds the median value for a field.<br>Note that the EQL definition of `MEDIAN` differs from the normal statistical definition when EQL is computing the median of an even number of numbers. That is, given an input relation containing `{1,2,3,4}`, the following query:<br>`RETURN results AS SELECT`<br>`  MEDIAN(a) AS med`<br>`GROUP`<br>produces the relation `{ <med:3> }`. According to the normal statistical definition, the statistical median of the set `{1, 2, 3, 4}` should be the mean of the two elements in the middle of the sorted set, or 2.5. |
| 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. |

## Hierarchy functions

EQL supports hierarchy functions on managed attributes.

You can filter by a descendant or an ancestor, or return a specific or relative level of the hierarchy. Managed attributes can be aliased in the `SELECT` statement and elsewhere.

The following are the related functions:

| Function | Description |
|---|---|
| `ANCESTOR(expr, int)` | Return the ancestor of the named attribute at the depth specified. Returns NULL if the requested depth is greater than the depth of the attribute value. The root is at depth 0. |
| `HIERARCHY_LEVEL(expr)` | Return the level of the named attribute as a number. The level is the number of values on the path from the root to it. The root is always level 0. |
| `IS_DESCENDANT(attribute, string)` | Include the record if the named attribute is the attribute specified or a descendant. If the attribute is not a member of the specified hierarchy, it is a compile-time error. If no attribute with the primary key in the attribute is found, it results in NULL. |

| Function | Description |
|---|---|
| `IS_ANCESTOR(attribute)` | Include the record if the named attribute is the attribute specified or an ancestor. If the attribute is not a member of the specified hierarchy, it is a compile-time error. If no attribute with the primary key in the attribute is found, it results in NULL. |
| `GET_LCA(attribute)` | A row function that returns the LCA (least common ancestor) of the two managed attributes. The two managed attributes should belong to same hierarchy. Otherwise, it is a compile-time error. |
| `LCA(attribute)` | An aggregation function that returns the LCA of the managed attributes in the specified attribute column. The LCA is the lowest point in a hierarchy that is an ancestor of all specified members. Any encountered NULL values are ignored by the function. |

In the first example, we filter by product category CAT_BIKES, and get all records assigned produce category CAT_BIKES or a descendant thereof:

```
RETURN example1 AS
SELECT
  ProductCategory AS ProductCategory,
  ANCESTOR(ProductCategory, 0) AS Ancestor
;
RETURN example2 AS
  ProductCategory AS ProductCategory,
  ANCESTOR(ProductCategory, HIERARCHY_LEVEL(ProductCategory)-1) AS Ancestor
WHERE
  IS_DESCENDANT(ProductCategory, 'CAT_BIKES')
```

In the second example, we want to return level 1 (one level below the root) of the Product Category hierarchy:

```
RETURN Results AS
SELECT
     ProductCategory AS PC,
     ANCESTOR(PC, 1) AS Ancestor
WHERE
     ANCESTOR(ProductCategory, 1) = 'CAT_BIKES'
GROUP BY PC
ORDER BY PC
```

In the third example, we want to return the direct ancestor of the Product Category hierarchy:

```
RETURN Results AS
SELECT
     ProductCategory AS PC,
     ANCESTOR(PC, HIERARCHY_LEVEL(PC) - 1) AS Parent
WHERE
     ANCESTOR(ProductCategory, 1) = 'CAT_BIKES'
GROUP BY PC
ORDER BY PC
```

In the second and third examples, we use `GROUP BY` to de-duplicate. In addition, note that even though we aliased `ProductCategory` AS `PC`, we cannot use the alias in the `WHERE` clause, because the alias does not become available until after `WHERE` clause executes.

**Note:** `GROUP BY` statements need to use the syntax `managedAttr:level`, rather than the `ANCESTOR` function, because you cannot group by an expression in EQL.

## Geocode functions

The geocode data type contains the longitude and latitude values that represent a geocode property.

The following are the related functions:

| Function | Description |
|---|---|
| `LATITUDE(mdex:geocode)` | Returns the latitude of a geocode as a floating-point number. |
| `LONGITUDE(mdex:geocode)` | Returns the longitude of a geocode as a floating-point number. |
| `DISTANCE(mdex:geocode, mdex:geocode)` | Returns the distance (in kilometers) between the two geocodes, using the haversine formula. |
| `TO_GEOCODE(mdex:float, mdex:float)` | Creates a geocode from the given latitude and longitude. |

The following example enables the display of a map with a pin for each location where a claim has been filed:

```
RETURN Result AS
SELECT
    LATITUDE(geo) AS Lat,
    LONGITUDE(geo) AS Lon,
    DISTANCE(geo, TO_GEOCODE(42.37, 71.13)) AS DistanceFromCambridge
WHERE
    DISTANCE(geo, TO_GEOCODE(42.37, 71.13)) BETWEEN 1 AND 10
```

**Note:** All distances are expressed in kilometers.

## Working with date and time values in EQL

EQL provides functions for working with `time`, `dateTime`, and `duration` data types.

EQL supports normal arithmetic operations between these data types.

All aggregation functions can be applied on these types except for `SUM`, which cannot be applied to `time` or `dateTime` types.

**Note:** In all cases, the internal representation of dates and times is on an abstract time line with no time zone. On this time line, all days are assumed to have exactly 86400 seconds. The system does not track, nor can it accommodate, leap seconds. This is equivalent to the SQL date, time, and timestamp data types that specify `WITHOUT TIMEZONE`. ISO 8601 ("Data elements and interchange formats - Information interchange - Representation of dates and times") recommends that, when communicating dates and times without timezone with other systems, they be represented using Zulu time, which is a synonym for GMT. Endeca Server conforms to this recommendation.

The following table summarizes the supported date and time functions:

| Function | Return Data Type | Purpose |
|---|---|---|
| CURRENT_TIMESTAMP<br>SYSTIMESTAMP | dateTime<br>dateTime | Constants representing the current date and time (at an arbitrary point during query evaluation) in GMT and server time zone, respectively. |
| CURRENT_DATE<br>SYSDATE | dateTime<br>dateTime | Constants representing current date (at an arbitrary point during query evaluation) in GMT and server time zone, respectively. |
| TO_TIME<br>TO_DATETIME<br>TO_DURATION | time<br>dateTime<br>duration | Constructs a timestamp representing time, date, or duration, using an expression. |
| EXTRACT | integer | Extracts a portion of a `dateTime` value, such as the day of the week or month of the year. |
| TRUNC | dateTime | Rounds a `dateTime` value down to a coarser granularity. |
| TO_TZ<br>FROM_TZ | dateTime<br>dateTime | Returns the given timestamp in a different time zone. |

The following table summarizes supported operations:

| Operation | Return Data Type |
|---|---|
| `time` (+\|-) `duration` | `time` |
| `dateTime` (+\|-) `duration` | `dateTime` |
| `time` - `time` | `duration` |
| `dateTime` - `dateTime` | `duration` |
| `duration` (+\|-) `duration` | `duration` |
| `duration` (*\|/) `double` | `duration` |
| `duration` /`duration` | `double` |

### Manipulating current date and time

EQL provides four constant keywords to obtain current date and time values. Values are obtained at an arbitrary point during query evaluation.

GMT time and date are independent of any daylight savings rules, while System time and date are subject to daylight savings rules.

| Keyword | Description |
|---|---|
| `CURRENT_TIMESTAMP` | Obtains current date and time in GMT. |
| `SYSTIMESTAMP` | Obtains current date and time in server time zone. |
| `CURRENT_DATE` | Obtains current date in GMT. |
| `SYSDATE` | Obtains system date in server time zone. |

**Note:** `CURRENT_DATE` and `SYSDATE` return dateTime data types where time fields are reset to zero.

The following example retrieves the average duration of service:

```
RETURN Example AS
SELECT AVG(CURRENT_DATE - DimEmployee_HireDate) AS DurationOfService
GROUP
```

### Constructing date and time values

EQL provides functions to construct a timestamp representing time, date, or duration using an expression.

If the expression is a string, it must be in a certain format. If the format is invalid or the value is out of range, it results in NULL.

<table>
<tr><th>Function</th><th>Description</th><th>Format</th></tr>
<tr><td>TO_TIME</td><td>Constructs a timestamp representing time.</td><td><Time String Format> ::= hh:mm:ss[.sss]((+|-) hh:mm |Z)</td></tr>
<tr><td>TO_DATETIME</td><td>Constructs a timestamp representing date and time.</td><td><DateTime String Format> ::= [-]YYYY-MM-DDT<Time String Format></td></tr>
</table>

<table>
<tr><th>Function</th><th>Description</th><th>Format</th></tr>
<tr><td>TO_DURATION</td><td>Constructs a timestamp representing duration.</td><td><Duration String Format> ::=<br>[-]P[<Days>][T(<Hours>[<Minutes>}[<Seconds>]|<br><Minutes>[<Seconds>]|<br><Seconds>)]<br><Days> ::= <Integer>D<br><Hours> ::= <Integer>H<br><Minutes> ::= <Integer>M<br><Seconds> ::= <Integer>[.<Integer>]S</td></tr>
</table>

As stated in the **Format** column above, TO_TIME and TO_DATETIME accept time zone offset. However, EQL does not store the offset value. Instead, it stores the value normalized to the GMT timezone.

The following table shows the output of several date and time expressions:

| Expression | Normalized value |
| --- | --- |
| TO_DATETIME('2012-03-21T16:00:00.000+02:00') | 2012-03-21T14:00:00.000Z |
| TO_DATETIME('2012-12-31T20:00:00.000-06:00') | 2013-01-01T02:00:00.000Z |
| TO_DATETIME('2012-06-15T20:00:00.000Z') | 2012-06-15T20:00:00.000Z |
| TO_TIME('23:00:00.000+03:00') | 20:00:00.000Z |
| TO_TIME('15:00:00.000-10:00') | 01:00:00.000Z |

### Timezone manipulation

EQL provides two functions to obtain the corresponding timestamp in different timezones.

EQL supports the standard IANA Time Zone database (*https://www.iana.org/time-zones*).

- TO_TZ Takes a timestamp in GMT, looks up the GMT offset for the specified timezone at that time in GMT, and returns a timestamp adjusted by that offset. If the specified timezone does not exist, the result is NULL. For example, TO_TZ(dateTime,'America/New_York') answers the question, "What time was it in America/New_York when it was dateTime in GMT?"
- FROM_TZ Takes a timestamp in the specified timezone, looks up the GMT offset for the specified timezone at that time, and returns a timestamp adjusted by that offset. If the specified timezone does not exist, the result is NULL. For example, FROM_TZ(dateTime,'EST') answers the question, "What time was it in GMT when it was dateTime in EST?"

The following table shows the results of several timezone expressions:

| Expression | Results |
|---|---|
| `TO_TZ(TO_DATETIME('2012-07-05T16:00:00.000Z'), 'America/New_York')` | 2012-07-05T12:00:00.000Z |
| `TO_TZ(TO_DATETIME('2012-01-05T16:00:00.000Z'), 'America/New_York')` | 2012-01-05T11:00:00.000Z |
| `FROM_TZ(TO_DATETIME('2012-07-05T16:00:00.000Z'), 'America/Los_Angeles')` | 2012-07-05T23:00:00.000Z |
| `FROM_TZ(TO_DATETIME('2012-01-05T16:00:00.000Z'), 'America/Los_Angeles')` | 2012-01-06T00:00:00.000Z |

#### Using EXTRACT to extract a portion of a dateTime value

The `EXTRACT` function extracts a portion of a `dateTime` value, such as the day of the week or month of the year. This can be useful in situations where the data must be filtered or grouped by a slice of its timestamps, for example to compute the total sales that occurred on any Monday.

The syntax of the `EXTRACT` function is:

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

| Date Time Unit | Range of Returned Values | Notes |
|---|---|---|
| `SECOND` | (0 - 59) | |
| `MINUTE` | (0 - 59) | |
| `HOUR` | (0 - 23) | |
| `DAY_OF_WEEK` | (1 - 7) | Returns the rank of the day within the week, where Sunday is 1. |
| `DAY_OF_MONTH (DATE)` | (1 -31) | |
| `DAY_OF_YEAR` | (1 - 365) | |
| `WEEK` | (1 - 53) | Returns the rank of the week in the year, where the first week starts on the first day of the year. |
| `MONTH` | (1 - 12) | |
| `QUARTER` | (1 - 4) | Quarters start in January, April, July, and October. |

| Date Time Unit | Range of Returned Values | Notes |
|---|---|---|
| YEAR | (-4713 - 9999) | |
| JULIAN_DAY_NUMBER | (0 - 5373484) | Returns the integral number of whole days between the timestamp and midnight, 24 November -4713. |

For example, the dateTime attribute TimeStamp has a value representing 10/13/2011 11:35:12.000. The following list shows the results of using the EXTRACT operator to extract each component of that value:

```
EXTRACT("TimeStamp", SECOND)              = 12
EXTRACT("TimeStamp", MINUTE)              = 35
EXTRACT("TimeStamp", HOUR)                = 11
EXTRACT("TimeStamp", DATE)                = 13
EXTRACT("TimeStamp", WEEK)                = 41
EXTRACT("TimeStamp", MONTH)               = 10
EXTRACT("TimeStamp", QUARTER)             = 4
EXTRACT("TimeStamp", YEAR)                = 2011
EXTRACT("TimeStamp", DAY_OF_WEEK)         = 5
EXTRACT("TimeStamp", DAY_OF_MONTH)        = 13
EXTRACT("TimeStamp", DAY_OF_YEAR)         = 286
EXTRACT("TimeStamp", JULIAN_DAY_NUMBER)   = 2455848
```

Here is a simple example of using this functionality. The following statement groups the total value of the Amount attribute by quarter, and for each quarter computes the total sales that occurred on a Monday (DAY_OF_WEEK=2):

```
RETURN Quarters AS
SELECT SUM(Amount) AS Total
       TRUNC(TimeStamp, QUARTER) AS Qtr
WHERE EXTRACT(TimeStamp,DAY_OF_WEEK) = 2
GROUP BY Qtr
```

The following example allows you to sort claims in buckets by age:

```
DEFINE ClaimsWithAge AS
SELECT

FLOOR((EXTRACT(TO_TZ(CURRENT_TIMESTAMP,claim_tz),JULIAN_DAY_NUMBER)-EXTRACT(TO_TZ(claim_ts,claim_tz),
JULIAN_DAY_NUMBER))/7) AS "AgeInWeeks",
    COUNT(1) AS "Count"
GROUP BY "AgeInWeeks"
HAVING "AgeInWeeks" < 2
ORDER BY "AgeInWeeks";

RETURN Result AS
SELECT
    CASE AgeInWeeks
        WHEN 0 THEN 'Past 7 Days'
        WHEN 1 THEN 'Prior 7 Days'
               ELSE 'Other'
        END
    AS "Label",
    "Count"
FROM ReviewsWithAge
```

### Using TRUNC to round down dateTime values

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

For example, this may be useful when you want to group your statement results data for each quarter using a `dateTime` attribute.

The syntax of the `TRUNC` function is:

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

**Note:** `WEEK` truncates to the nearest previous Sunday.

For example, the `dateTime` attribute TimeStamp has a value representing 10/13/2011 11:35:12.000. The list below shows the results of using the `TRUNC` operator to round the TimeStamp value at each level of granularity. The values are displayed here in a format that is easier to read—the actual values would use the standard Endeca `dateTime` format.

```
TRUNC("TimeStamp", SECOND)            = 10/13/2011 11:35:12.000
TRUNC("TimeStamp", MINUTE)            = 10/13/2011 11:35:00.000
TRUNC("TimeStamp", HOUR)              = 10/13/2011 11:00:00.000
TRUNC("TimeStamp", DATE)              = 10/13/2011 00:00:00.000
TRUNC("TimeStamp", WEEK)              = 10/09/2011 00:00:00.000
TRUNC("TimeStamp", MONTH)             = 10/01/2011 00:00:00.000
TRUNC("TimeStamp", QUARTER)           = 10/01/2011 00:00:00.000
TRUNC("TimeStamp", YEAR)              = 01/01/2011 00:00:00.000
TRUNC("TimeStamp", DAY_OF_WEEK)       = 10/13/2011 00:00:00:000
TRUNC("TimeStamp", DAY_OF_MONTH)      = 10/13/2011 00:00:00:000
TRUNC("TimeStamp", DAY_OF_YEAR)       = 10/13/2011 00:00:00:000
TRUNC("TimeStamp", JULIAN_DAY_NUMBER) = 10/13/2011 00:00:00:000
```

Here is a simple example of using this functionality. In the following statement, the total value for the Amount attribute is grouped by quarter. The quarter is obtained by using the `TRUNC` operation on the TimeStamp attribute:

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

### Using arithmetic operations on date and time values

In addition to using the `TRUNC` and `EXTRACT` functions, you also can use normal arithmetic operations with date and time values.

The following are the supported operations:

- Add or subtract a duration to or from a time or a dateTime to obtain a new time or dateTime.
- Subtract two times or dateTimes to obtain a duration.
- Add or subtract two durations to obtain a new duration.
- Multiply or divide a duration by a double number.
- Divide a duration by a duration.

The following table shows the results of several arithmetic operations on date and time values:

| Expression | Results |
|---|---|
| `2012-10-05T00:00:00.000Z` + `P30D` | 2012-11-04T00:00:00.000Z |
| `2012-10-05T00:00:00.000Z` - `PT01M` | 2012-10-04T23:59:00.000Z |
| `23:00:00.000Z` + `PT02H` | 01:00:00.00 |
| `20:00:00.000Z` - `PT02S` | 19:59:58.000Z |
| `2012-01-01T00:00:00.000Z` - `2012-12-31T00:00:00.000Z` | -P365DT0H0M0.000S |
| `23:15:00.000Z` - `20:12:30.500Z` | P0DT3H2M29.500S |
| `P1500DT0H0M0.000S` - `P500DT0H0M0.000S` | P1000DT0H0M0.000S |
| `P1DT0H30M0.500S` * `2.5` | P2DT13H15M1.250S |
| `P1DT0H30M0.225S` / `2` | P0DT12H15M0.112S |
| `P5DT12H00M0.000S` / `P1DT0H00M0.000S` | 5.5 |

### Arithmetic operators

EQL supports arithmetic operators for addition, subtraction, multiplication, and division.

The syntax is as follows:

```
<expr> {+, -, *, /} <expr>
```

Each arithmetic operator has a corresponding numeric function. For information on order of operations, see *Operator precedence rules on page 19*.

### Boolean operators

EQL supports the Boolean operators `AND`, `OR`, and `NOT`.

For information on order of operations, see *Operator precedence rules on page 19*.

## Using EQL results to compose follow-on queries

You can select a value in an EQL result and use it to compose a follow-on query.

This enables users to interact with EQL results through a chart or a graph to compose follow-on queries. For example, when viewing a chart of year-to-date sales by country, a user might select a specific country for drill-down.

EQL is specifically designed to support this kind of follow-on query.

If, in the above example, the user selects the country United States, then the follow-on query should examine only sales of products in the United States. To filter to these items, a `WHERE` clause like the following can be added:

```
WHERE DimGeography_CountryRegionName = 'United States'
```

For attributes with types other than string, a conversion is necessary to use the string representation of the value returned by EQL. For an integer attribute, like `DimDate_CalendarYear`, the string representation of the value must be converted to an integer for filtering, as follows:

```
WHERE DimDate_CalendarYear = TO_INTEGER('2006').
```

EQL provides conversions for all non-string data types:

- `TO_BOOLEAN()`
- `TO_DATETIME()`
- `TO_DOUBLE()`
- `TO_DURATION()`
- `TO_GEOCODE()`
- `TO_INTEGER()`
- `TO_TIME()`

Each of these accepts the string representation of values produced by the Endeca Server. Note that, for `mdex:string` attributes (including managed attributes), no conversion is necessary.

To determine which conversion function to use, EQL results are accompanied by attribute metadata that describes both the type of the attribute, and, for managed attributes, any associated hierarchy.

### Filtering to a node in a hierarchy

When filtering to a node in a hierarchy, such as ProductCategory, users typically want to filter to records that are tagged with a particular value or any of its descendants. For example, if a user drills into Accessories, filtering to records tagged with Accessories will return no results. However, filtering with:

```
WHERE IS_DESCENDANT(ProductCategory, 'Accessories')
```

produces the desired result of filtering to records tagged with Accessories or any descendent thereof.

## Using AS expressions to calculate derived attributes

EQL statements typically use expressions to compute one or more derived attributes.

Each aggregation operation can declare an arbitrary set of named expressions, sometimes referred to as derived attributes, using `SELECT AS` syntax. These expressions represent aggregate analytic functions that are computed for each aggregated record in the statement result.

**Important:** Derived attribute names must be NCName-compliant. They cannot contain spaces or special characters. For example, the following statement would not be valid:

```
RETURN price AS SELECT AVG(Price) AS "Average Price"
```

The space would have to be removed:

```
RETURN price AS SELECT AVG(Price) AS "AveragePrice"
```

The NCName format is defined in the W3C document Namespaces in XML 1.0 (Second Edition), located at this URL: *http://www.w3.org/TR/REC-xml-names/*.

## COALESCE

The `COALESCE` expression allows for user-specified NULL-handling. It is often used to fill in missing values in dirty data.

It has a function-like syntax, but can take unlimited arguments, for example: `COALESCE(a, b, c, x, y, z)`.

You can use the `COALESCE` expression 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` must all be of the same type, with the following exceptions:
  - Integers with doubles (resulting in doubles)
  - Strings with managed attributes (resulting in managed attributes)

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 without aggregation, for example:

```
SELECT COALESCE(price, 0) AS price_or_zero WHERE ...
```

## CASE

`CASE` expressions allow conditional processing in EQL, allowing you to make decisions at query time.

The syntax of the `CASE` expression, which conforms to the SQL standard, is:

```
CASE
     WHEN <Boolean expression> THEN <expression>
     [WHEN <Boolean expression> THEN <expression>]*
     [ELSE expression]
END
```

`CASE` expressions must include at least one `WHEN` expression. The first `WHEN` expression with a `TRUE` condition is the one selected. `NULL` is not `TRUE`. The optional `ELSE` clause must always come at the end of the `CASE` statement and is equivalent to `WHEN TRUE THEN`. If no condition matches, the result is `NULL`.

In this example, division by non-positive integers is avoided:

```
CASE
     WHEN y < 0 THEN x / (0 - y)
     WHEN y > 0 THEN x / y
     ELSE 0
END
```

In this example, records are categorized as Recent or Old:

```
RETURN Result AS
SELECT
  CASE
    WHEN (Days < 7) THEN 'Recent'
    ELSE ‘Old’
  END AS Age
```

The following example groups all records by class and computes the following:

- The minimum DealerPrice of all records in class H.
- The minimum ListPrice of all records in class M.
- The minimum StandardCost of all other records (called class L).

```
RETURN CaseExample AS SELECT
      CASE
        WHEN Class = 'H' THEN MIN(DealerPrice)
        WHEN Class = 'M' THEN MIN(ListPrice)
        ELSE MIN(StandardCost)
      END
AS value
GROUP BY Class
```

## Using inter-statement references

In EQL, you can define statements and then refer to these statements from other statements.

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

Expressions also can use values from other computed statements. This is often useful when coarser subtotals are required for computing analytics within a finer-grained bucket.

For example, when computing the percent contribution for each sales representative in a given year, you must also calculate the overall total for the year. You can use inter-statement references to create these types of queries.

### Syntax for inter-statement references

The syntax for an inter-statement reference is:

```
<LookupExpr>   ::=  <statement name>[<LookupList>].<attribute name>
<LookupList>   ::=  <empty>
               ::=  <SimpleExpr> [,<LookupList>]
```

The square brackets are used to identify the record set and grouping attribute, and the dot is used to identify the field.

### Referencing a value from another statement

For 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.

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

RETURN ProductPcts AS
SELECT
```

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

The first statement computes the total product sales for each region. The next statement then uses the RegionTotals results to determine the percentage for each region, making use of the inter-statement reference syntax.

- The bracket operator indicates to reference the RegionTotals result that has a group-by value equal to the ProductPcts value for the Region attribute.
- The dot operator indicates to reference the Total field in the specified RegionTotals record.

### Computing percentage of sales

This example computes for each quarter the percentage of sales for each product type.

This query requires calculating information in one statement in order to use it in another statement.

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. The calculations for quarter/product pairs can then 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
```

## IN

`IN` expressions perform a membership test.

`IN` expressions address use cases where you want to identify a set of interest, and then filter to records with attributes that are in or out of that set. They are useful in conjunction with `HAVING` and `PAGE` expressions.

The syntax is as follows:

```
[Attr1, Attr2, …] IN StatementName
```

The example below helps answer the questions, "Which products do my highest value customers buy?" and "What is my total spend with suppliers from which I purchase my highest spend commodities?"

```
DEFINE HighValueCust AS SELECT
  SUM(SalesAmount) AS Value
GROUP BY CustId
HAVING Value>10000 ;

RETURN Top_HVC_Products AS SELECT
  COUNT(1) AS NumSales
WHERE [CustId] IN HighValueCust
GROUP BY ProductName
ORDER BY NumSales DESC
PAGE(0,10)
```

## LOOKUP

A `LOOKUP` expression is a simple form of join. It treats the result of a prior statement as a lookup table.

Its syntax is as follows:

```
<statement>[<expression list>].<attribute>
```

The expression list corresponds to the grouping attributes of the specified statement. If any of the expressions in the list is `NULL`, the result is `NULL`.

Lookup attributes refer to `GROUP BY`s of the target statement, in order. Computed lookup of indexed values is allowed, which means you can look up related information, such as total sales from the prior year, as shown in the following example:

```
DEFINE YearTotals AS SELECT
  SUM(SalesAmount) AS Total
GROUP BY Year ;

RETURN AnnualCategoryPcts AS SELECT
  SUM(SalesAmount) AS Total,
  Total/YearTotals[Year].Total AS Pct
GROUP BY Year, Category ;

RETURN YoY AS SELECT
  YearTotals[Year].Total AS Total,
  YearTotals[Year-1].Total AS Prior,
  (Total-Prior)/Prior AS PctChange
GROUP BY Year
```

Chapter 3

# EQL Use Cases

This section introduces a number of use case examples for various business scenarios. The examples in this section are not based on a single data schema.



## Re-normalization

Re-normalization is important in denormalized data models in the Endeca Server, as well as when analyzing multi-value attributes.

In the Quick Start data, Employees were de-normalized onto Transactions, as shown in the following example:

| | |
|---|---|
| DimEmployee_FullName: | Tsvi Michael Reiter |
| DimEmployee_HireDate: | 2005-07-01T04:00:00.000Z |
| DimEmployee_Title: | Sales Representative |
| FactSales_RecordSpec: | SO49122-2 |
| FactSales_SalesAmount: | 939.588 |

### Incorrect

The following EQL code double-counts the tenure of Employees with multiple transactions:

```
RETURN AvgTenure AS SELECT
   AVG(CURRENT_DATE - DimEmployee_HireDate) AS AvgTenure GROUP BY DimEmployee_Title
```

### Correct

In this example, you re-normalize each Employee, and then operate over them using `FROM`:

```
DEFINE Employees AS SELECT
   DimEmployee_HireDate AS DimEmployee_HireDate,
   DimEmployee_Title AS DimEmployee_Title GROUP BY DimEmployee_EmployeeKey;

RETURN AvgTenure AS SELECT
   AVG(CURRENT_DATE - DimEmployee_HireDate) AS AvgTenure FROM Employees GROUP BY DimEmployee_Title
```

## Grouping by range buckets

To create value range buckets, divide the records by the bucket size, and then use `FLOOR` or `CEIL` if needed to round to the nearest integer.

The following examples group sales into buckets by amount:

```
/**
  * This groups results into buckets by amount,
  * rounded to the nearest 1000.
  */
RETURN Results AS
SELECT
   ROUND(FactSales_SalesAmount, -3) AS Bucket,
   COUNT(1) AS "Count"
GROUP BY Bucket
```

```
/**
  * This groups results into buckets by amount,
  * truncated to the next-lower 1000.
  */
RETURN Results AS
SELECT
   FLOOR(FactSales_SalesAmount/1000)*1000 AS Bucket,
   COUNT(1) AS "Count"
GROUP BY Bucket
```

A similar effect can be achieved with `ROUND`, but the set of buckets is different:

- `FLOOR(900/1000)` = `0`
- `ROUND(900,-3)` = `1000`

In the following example, records are grouped into a fixed number of buckets:

```
DEFINE ValueRange AS SELECT
   COUNT(1) AS "Count"
GROUP BY SalesAmount
HAVING SalesAmount > 1.0
   AND SalesAmount < 10000.0;

RETURN Buckets AS SELECT
   SUM("Count") AS "Count",
   FLOOR((SalesAmount - 1)/999.0) AS Bucket
```

```
FROM ValueRange
GROUP BY Bucket
ORDER BY Bucket
```

## Manipulating records in a dynamically computed range value

The following scenario describes how to manipulate records in a dynamically computed range value.

In the following example:

- Use `GROUP` to calculate a range of interest.
- Empty `LOOKUP` to get the range of interest into the desired expression.
- Use subtraction and `HAVING` to enable filtering by a dynamic value (instead of a static constant, as required by `WHERE`).

```
DEFINE CustomerTotals AS SELECT
    SUM(SalesAmount) AS Total
GROUP BY CustomerKey ;

DEFINE Range AS SELECT
    MAX(Total) AS MaxVal,
    MIN(Total) AS MinVal,
    ((MaxVal-MinVal)/10) AS Decile,
    MinVal + (Decile*9) AS Top10Pct
FROM CustomerTotals GROUP ;

RETURN Result AS SELECT
    SUM(SalesAmount) AS Total,
    Total-Range[].Top10Pct AS Diff
GROUP BY CustomerKey
HAVING Diff>0
```

## Grouping data into quartiles

EQL allows you to group your data into quartiles.

The following example demonstrates how to group data into four roughly equal-sized buckets.

```
/* This finds quartiles in the range
  * of ProductSubCategory, arranged by
  * total sales. Adjust the grouping
  * attribute and metric to your use case.
  */
DEFINE Input AS SELECT
   ProductSubcategoryName AS Key,
   SUM(FactSales_SalesAmount) AS Metric
GROUP BY Key
ORDER BY Metric;

DEFINE Quartile1Records AS SELECT
   Key AS Key,
   Metric AS Metric
FROM Input
ORDER BY Metric
PAGE(0, 25) PERCENT;
```

```
/* Using MAX(Metric) as the Quartile boundary isn't quite
  * right: if the boundary falls between two records, the
  * quartile is the average of the values on those two records.
  * But this gives the right groupings.
  */
DEFINE Quartile1 AS SELECT
   MAX(Metric) AS Quartile,
   SUM(Metric) AS Metric /* ...or any other aggregate */
FROM Quartile1Records
GROUP;

DEFINE Quartile2Records AS SELECT
   Key AS Key,
   Metric AS Metric
FROM Input
ORDER BY Metric
PAGE(25, 25) PERCENT;

DEFINE Quartile2 AS SELECT
   MAX(Metric) AS Quartile,
   SUM(Metric) AS Metric
FROM Quartile2Records
GROUP;

DEFINE Quartile3Records AS SELECT
   Key AS Key,
   Metric AS Metric
FROM Input
ORDER BY Metric
PAGE(50, 25) PERCENT;
```

```
DEFINE Quartile3 AS SELECT
   MAX(Metric) AS Quartile,
   SUM(Metric) AS Metric
FROM Quartile3Records
GROUP;

DEFINE Quartile4Records AS SELECT
   Key AS Key,
   Metric AS Metric
FROM Input
ORDER BY Metric
PAGE(75, 25) PERCENT;

DEFINE Quartile4 AS SELECT
   MAX(Metric) AS Quartile,
   SUM(Metric) AS Metric
FROM Quartile4Records
GROUP;

/**
  * The technical definition of "Quartile" is
  * the values that segment the data into four
  * roughly equal groups. Here, we return not
  * just the Quartiles, but the metric aggregated
  * over the records within the groups defined
  * by the Quartiles.
  */
RETURN Quartiles AS
SELECT
   Quartile AS Quartile1,
   Metric AS Quartile1Metric,
   Quartile2[].Quartile AS Quartile2,
   Quartile2[].Metric AS Quartile2Metric,
   Quartile3[].Quartile AS Quartile3,
   Quartile3[].Metric AS Quartile3Metric,
   Quartile4[].Quartile AS Quartile4,
   Quartile4[].Metric AS Quartile4Metric FROM Quartile1;
```

# Combining multiple sparse fields into one

EQL allows you to combine multiple sparse fields into a single field.

In the example below, we use the `AVG` and `COALESCE` functions to combine the `leasePayment` and `loanPayment` fields into a single `avgPayment` field.

| ID | Make | Model | Type | leasePayment | loanPayment |
|---|---|---|---|---|---|
| 1 | Audi | A4 | lease | 380 | |
| 2 | Audi | A4 | loan | | 600 |
| 3 | BMW | 325 | lease | 420 | |
| 4 | BMW | 325 | loan | | 700 |

```
RETURN Result AS SELECT
  AVG(COALESCE(loanPayment,leasePayment))
    AS avgPayment
FROM CombinedColumns
GROUP BY make
```

# Counting multi-assign terms

Take care when counting multi-assign terms to ensure you capture all assignments.

The first, incorrect example only counts a single arbitrary term assignment per record scanned:

```
RETURN TermCounts AS SELECT
    COUNTDISTINCT(Term) as NumTerms, /* wrong; term is de-multi-assigned
    * prior to COUNTDISTINCT */
    COUNT(Term) as NumAssignments
GROUP BY Category
```

The second, correct example uses a `SUM` of `COUNTs` pattern. This pattern can be used any time where it is useful to first produce partial `COUNTs` and then add them up to get the total `COUNT`.

```
DEFINE Terms AS SELECT
    COUNT(1) AS Assignments
GROUP BY Term, Category ;

RETURN TermCounts AS SELECT
    COUNTDISTINCT(Term) as NumTerms,
    SUM(Assignments) AS NumAssignments
FROM Terms
GROUP BY Category
```

# Joining data from different types of records

You can use EQL to join data from different types of records.

Use lookups against `AllBaseRecords` to avoid eliminating all records of a secondary type when navigation refinements are selected from an attribute only associated with the primary record type.

In the following example, the following types of records are joined:

**Record type 1**

`RecordType:` `Review`

`Rating:` `4`

`ProductId:` `Drill-X15`

`Text:` `This is a great product...`

**Record type 2**

`RecordType:` `Transaction`

`SalesAmount:` `49.99`

`ProductId:` `Drill-X15`

`...`

```
DEFINE Ratings AS SELECT
    AVG(Rating) AS AvScore
FROM AllBaseRecords
WHERE RecordType = 'Review'
GROUP BY ProductId ;

RETURN TopProducts AS SELECT
   SUM(SalesAmount) AS TotalSales,
   Ratings[ProductId].AvScore AS AvScore
WHERE RecordType = 'Transaction'
GROUP BY ProductId
ORDER BY TotalSales DESC
PAGE(0,10)
```

# Joining on hierarchy

The following example shows a transitive join on hierarchy.

This query returns the number of reports in each manager's `Org`. (`Org` is a managed attribute representing organizational structure.)

```
RETURN SELECT
   COUNT(1) AS TotalMembers,
   manager.Org AS Org
FROM People manager
   JOIN People report
   ON IS_ANCESTOR(manager.Org, report.Org)
GROUP BY Org
```

# Linear regressions in EQL

Using the syntax described in this topic, you can produce linear regressions in EQL.

Using the following data set:

| ID | X | Y |
|---|---|---|
| 1 | 60 | 3.1 |
| 2 | 61 | 3.6 |
| 3 | 62 | 3.8 |
| 4 | 63 | 4 |
| 5 | 65 | 4.1 |

The following simple formulation:

y = A + Bx

Can be expressed in EQL as:

```
RETURN Regression AS SELECT
    COUNT(ID) AS N,
    SUM(X) AS sumX,
    SUM(Y) AS sumY,
    SUM(X*Y) AS sumXY,
    SUM(X*X) AS sumX2,
    ((N*sumXY)-(sumX*sumY)) /
     ((N*sumX2)-(sumX*sumX)) AS B,
    (sumY-(B*sumX))/N AS A
GROUP
```

With the result:

| N | sumX | sumY | sumXY | sumX2 | B | A |
|---|---|---|---|---|---|---|
| 5 | 311.000000 | 18.600000 | 1159.700000 | 19359.000000 | 0.187838 | -7.963514 |

## Using the regression results

For y = A + Bx:

```
DEFINE Regression AS SELECT
    COUNT(ID) AS N,
    SUM(X) AS sumX,
    SUM(Y) AS sumY,
    SUM(X*Y) AS sumXY,
    SUM(X*X) AS sumX2,
    ((N*sumXY)-(sumX*sumY)) /
    ((N*sumX2)-(sumX*sumX)) AS B,
    (sumY-(B*sumX))/N AS A
GROUP

RETURN Results AS SELECT
```

```
    Y AS Y,  X AS X,   Regression[].A + Regression[].B * X AS Projection
...
```

As a final step in the example above, you would need to `PAGE` or `GROUP` what could be a very large number of results.

# Using an IN filter for pie chart segmentation

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
```

# Running sum

A running (or cumulative) sum calculation can be useful in warranty scenarios.

```
/* This selects the total sales in the
  * 12 most recent months.
  */
DEFINE Input AS SELECT
   DimDate_CalendarYear AS "Year",
   DimDate_MonthNumberOfYear AS "Month",
   SUM(FactSales_SalesAmount) AS TotalSales GROUP BY "Year", "Month"
ORDER BY "Year" DESC, "Month" DESC
PAGE(0, 12);

RETURN CumulativeSum AS SELECT
   one."Year" AS "Year",
   one."Month" AS "Month",
   SUM(many.TotalSales) AS TotalSales
FROM Input one JOIN Input many
ON ((one."Year" > many."Year") OR
     (one."Year" = many."Year" AND
      one."Month" >= many."Month")
     )
GROUP BY "Year", "Month"
ORDER BY "Year", "Month"
```

# Query by age

In this example, records are tagged with a Date attribute on initial ingest. No updates are necessary.

```
RETURN Result AS
```

```
SELECT
  EXTRACT(CURRENT_DATE,
      JULIAN_DAY_NUMBER) -
    EXTRACT(Date, JULIAN_DAY_NUMBER)
    AS AgeInDays
HAVING (AgeInDays < 30)
```

# Calculating percent change between most recent month and previous month

The following example finds the most recent month in the data that matches the current filters, and compares it to the prior month, again in the data that matches the current filters.

```
/* This computes the percent change between the most
  * recent month in the current nav state, compared to the prior
  * month in the nav state. Note that, if there's only
  * one month represented in the nav state, this will return NULL.
  */
DEFINE Input AS SELECT
   DimDate_CalendarYear AS "Year",
   DimDate_MonthNumberOfYear AS "Month",
   DimDate_CalendarYear * 12 + DimDate_MonthNumberOfYear AS OrdinalMonth,
   SUM(FactSales_SalesAmount) AS TotalSales GROUP BY OrdinalMonth;

RETURN Result AS SELECT
   "Year" AS "Year",
   "Month" AS "Month",
   TotalSales AS TotalSales,
   Input[OrdinalMonth - 1].TotalSales AS PriorMonthSales,
   100 * (TotalSales - PriorMonthSales)
/ PriorMonthSales AS PercentChange FROM Input ORDER BY "Year" DESC, "Month" DESC PAGE(0, 1)
```

## Chapter 4
# EQL Best Practices

This section discusses ways to maximize your EQL query performance.



## Controlling input size

The size of the input for a statement can have a big impact on the evaluation time of the query.

The input for a statement is defined by the `FROM` clause. If no `FROM` clause is provided, the input defaults to the `NavStateRecords`. When possible, use an already completed result from another statement, instead of using corpus records, to avoid inputting unnecessary records.

Consider the following queries. In the first query, the input to each statement is of a size on the order of the navigation state. In the first two statements, Sums and Totals, the data is aggregated at two levels of granularity. In the last statement, the data set is accessed again for the sole purpose of identifying the month/year combinations that are present in the data. The computations of interest are derived from previously-computed results.

```
DEFINE Sums AS SELECT
  SUM(a) AS MonthlyTotal
GROUP BY month,year;

DEFINE Totals AS SELECT
  SUM(a) AS YearlyTotal
GROUP BY year;

DEFINE Result AS SELECT
  Sums[month,year].MonthlyTotal AS MonthlyTotal,
  Sums[month,year].MonthlyTotal/Totals[year].YearlyTotal AS Fraction
GROUP BY month,year
```

In the following rewrite of the query, the index is accessed only once. The first statement accesses the index to compute the monthly totals. The second statement has been modified to compute yearly totals using the results of the first statement. Assuming that there are many records per month, the savings could be multiple orders of magnitude. Finally, the last statement has also been modified to use the results of the first statement. The first statement has already identified all of the valid month/year combinations in the data set. Rather than accessing the broader data set (possibly millions of records) just to identify the valid combinations, the month/year pairs are read from the much smaller (probably several dozen records) previous result.

```
DEFINE Sums AS SELECT
  SUM(a) AS MonthlyTotal
```

```
GROUP BY month,year;

DEFINE Totals AS SELECT
  SUM(MonthlyTotal) AS YearlyTotal
FROM Sums
GROUP year;

DEFINE Result AS SELECT
  MonthlyTotal AS MonthlyTotal,
  MonthlyTotal/Totals[year].YearlyTotal AS Fraction
FROM Sums
```

### Defining constants independent of data set size

A common practice is to define constants for a query through a single group, as shown in the first query below. Note that the input for this query is the entire navigation state, even though nothing from the input is used. Since none of the input is actually needed, restrict the input to the smallest size possible with a very restrictive filter, such as the one shown in the second example.

```
DEFINE Constants AS SELECT
  500 AS DefaultQuota
GROUP
```

```
DEFINE Constants AS SELECT
  500 AS DefaultQuota
WHERE "mdex-property_Key" IS NOT NULL
GROUP
```

## Filtering as early as possible

Filtering out rows as soon as possible improves query latency because it reduces the amount of data that must be tracked through the evaluator.

Consider the following two versions of a query. The first form of the query first groups records by `g`, passes each group through the filter `(b < 10)`, and then accumulates the records that remain. The input records are not filtered, and the grouping operation must operate on all input records.

```
RETURN Result AS SELECT
  SUM(a) WHERE (b < 10) AS sum_a_blt10
GROUP BY g
```

The second form of the query filters the input (with the `WHERE` clause) before the records are passed to the grouping operation. Thus the grouping operation must group only those records of interest to the query. By eliminating records that are not of interest sooner, evaluation will be faster.

```
RETURN Results AS SELECT
  SUM(a) AS sum_a_blt10,
WHERE (b < 10)
GROUP BY g
```

Another example of filtering records early is illustrated with the following pair of queries. Recall that a `WHERE` clauses filters input records and a `HAVING` clause filters output records. The first query computes the sum for all values of `g` and (after performing all of that computation) throws away all results that do not meet the condition `(g < 10)`.

```
RETURN Result AS SELECT
  SUM(a) AS sum_a
GROUP BY g
```

```
HAVING g < 10
```

The second query, on the other hand, first filters the input records to only those in the interesting groups. It then aggregates only those interesting groups.

```
RETURN Result AS SELECT
SUM(a) AS sum_a
WHERE g < 10
GROUP BY g
```

# Controlling join size

Joins can cause the Endeca Server to grow beyond available RAM. Going beyond the scale capabilities will cause very, very large materializations, intense memory pressure, and can result in an unresponsive Endeca Server.

# Additional tips

This topic contains additional tips for working effectively with EQL.

- String manipulations are unsupported in EQL. Therefore, ensure you prepare string values for query purposes in the data ingest stage.
- Normalize information to avoid double counting or summing, as well as to prevent the production of arbitrary values with multi-assign attributes.
- Use a common case (upper case) for attribute string values when sharing attributes between data sources.
- Name each `DEFINE` statement something meaningful so that others reading your work can make sense of what your logic is.
- Use paging in `DEFINE` statements to reduce the number of records returned.
- When using `CASE` statements, bear in mind that all conditions and expressions are always evaluated, even though only one is returned. If an expression is repeated across multiple `WHEN` clauses of a `CASE` expression, it is best to factor the computation of that expression out into a separate `SELECT`, then re-use it.

# Index