Oracle® Endeca Server

EQL Guide

Version 7.5.1.1 • May 2013

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

The following conventions are used in this document.

Typographic conventions

This table describes the typographic conventions used when formatting text in this document.

Typeface	Meaning
User Interface Elements	This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields.
`Code Sample`	This formatting is used for sample code phrases within a paragraph.
Variable	This formatting is used for variable values. For variables within a code sample, the formatting is ***`Variable`***.
`File Path`	This formatting is used for file names and paths.

Symbol conventions

This table describes the symbol conventions used in this document.

Symbol	Description	Example	Meaning
>	The right angle bracket, or greater-than sign, indicates menu item selections in a graphic user interface.	File > New > Project	From the File menu, choose New, then from the New submenu, choose Project.

Path variable conventions

This table describes the path variable conventions used in this document.

Path variable	Meaning
`$MW_HOME`	Indicates the absolute path to your Oracle Middleware home directory, which is the root directory for your WebLogic installation.
`$DOMAIN_HOME`	Indicates the absolute path to your WebLogic domain home directory. For example, if `endeca_server_domain` is the name of your WebLogic domain, then the `$DOMAIN_HOME` value would be the `$MW_HOME/user_projects/domains/endeca_server_domain` directory.
`$ENDECA_HOME`	Indicates the absolute path to your Oracle Endeca Server home directory, which is the root directory for your Endeca Server installation.

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 chapter introduces the Endeca Query Language (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 if it is between statements (the semi-colon is optional on the last statement). The statement also 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 optional when selecting an attribute verbatim; statements using expressions require an `AS` alias. Aliasing 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.
  - `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.

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.

Syntax conventions used in this guide

The syntax descriptions in this guide use the following conventions:

Convention	Meaning	Example
Square brackets [ ]	Optional	`FROM <statementKey> [alias]`
Asterisk *	May be repeated	`[, JOIN statement [alias] ON <Boolean expression>]*`

Convention	Meaning	Example
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 that EQL comments cannot be nested.

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` clauses) and arguments to aggregations.

3. It maps source records to result records and computes aggregations.
4. It finishes computing `SELECT` clauses.
5. It filters result records according to the `HAVING` clause.
6. It orders result records.
7. It applies paging to the results.

EQL requests in the Conversation Service

A request made with the Conversation Web Service can include statements in EQL.

The Conversation Service's `LQLConfig` type lets you make queries using EQL statements.

Consider the following EQL statement:

```
RETURN SalesTransactions AS SELECT SUM(FactSales_SalesAmount)
WHERE (DimDate_FiscalYear=2008) AS Sales2008,
SUM(FactSales_SalesAmount)
WHERE (DimDate_FiscalYear=2007) AS Sales2007,
((Sales2008-Sales2007)/Sales2007 * 100) AS pctChange,
COUNTDISTINCT(FactSales_SalesOrderNumber)
AS TransactionCount
GROUP
```

To send it for processing to the Oracle Endeca Server, use the `LQLConfig` type of `ContentElementConfig`, including the statement inside the `LQLQueryString` element, as in this example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns="http://www.endeca.com/MDEX/conversation/2/0"
  xmlns:typ="http://www.endeca.com/MDEX/lql_parser/types">
<soapenv:Header/>
<soapenv:Body>
 <ns:Request>
 <ns:Language>en</ns:Language>
  <ns:State/>
   <ns:ContentElementConfig Id="LQLConfig" xsi:type="ns:LQLConfig"
    HandlerNamespace="http://www.endeca.com/MDEX/conversation/1/0"
    HandlerFunction="LQLHandler"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <ns:LQLQueryString>
      RETURN statement AS SELECT SUM(FactSales_SalesAmount)
      WHERE (DimDate_FiscalYear=2008) AS Sales2008,
      SUM(FactSales_SalesAmount) WHERE (DimDate_FiscalYear=2007) AS Sales2007,
      ((Sales2008-Sales2007)/Sales2007 * 100) AS pctChange,
      countDistinct(FactSales_SalesOrderNumber)
      AS TransactionCount
      group
     </ns:LQLQueryString>
    </ns:ContentElementConfig>
   </ns:Request>
  </soapenv:Body>
</soapenv:Envelope>
```

The contents of the `LQLQueryString` element must be a valid EQL statement.

The `HandlerFunction` that supports processing of the `LQLConfig` is `LQLHandler`.

The following abbreviated response returned from the Conversation Web Service contains the calculated results of the EQL statements:

```
<cs:ContentElement xsi:type="cs:LQL" Id="LQLConfig"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
<cs:ResultRecords NumRecords="1" Name="SalesTransactions">
 <cs:DimensionHierarchy/>
  <cs:AttributeMetadata name="Sales2007" type="mdex:double"/>
  <cs:AttributeMetadata name="Sales2008" type="mdex:double"/>
  <cs:AttributeMetadata name="TransactionCount" type="mdex:long"/>
  <cs:AttributeMetadata name="pctChange" type="mdex:double"/>
     <cs:Record>
       <Sales2007 type="mdex:double">2.79216705182E7</Sales2007>
       <Sales2008 type="mdex:double">3.62404846965997E7</Sales2008>
       <TransactionCount type="mdex:long">3796</TransactionCount>
       <pctChange type="mdex:double">29.793397114178</pctChange>
    </cs:Record>
  </cs:ResultRecords>
</cs:ContentElement>
```

Note: This example shows only one of the ways for using EQL statements in Conversation Web Service requests. Typically, requests also include `State` and `Operator` elements that define the navigation state. In your EQL statement, you can select from this navigation state using the From clause.

Language ID for parsing error messages

The `Request` complex type has an optional `Language` element that sets the language for error messages that result from EQL parsing. The supported languages and their corresponding language IDs are:

- Chinese (simplified): `zh_CN`
- Chinese (traditional): `zh_TW`
- English: `en`
- French: `fr`
- German: `de`
- Italian: `it`
- Japanese: `ja`
- Korean : `ko`
- Portuguese: `pt`
- Spanish: `es`

If a language ID is not specified, then `en` (English) is the default.

The `LQLQueryString` example above shows where in the request you would specify the `Language` element for EQL parsing error messages.

EQL reserved keywords

EQL reserves certain keywords for its exclusive use.

Reserved keywords

Reserved keywords cannot be used in EQL statements as identifiers, unless they are delimited by double quotation marks. For example, this EQL snippet uses the `YEAR` and `MONTH` reserved keywords as delimited identifiers:

```
DEFINE Input AS SELECT
   DimDate_CalendarYear AS "Year",
   DimDate_MonthNumberOfYear AS "Month",
   ...
```

However, as a rule of thumb it is recommended that you do not name any identifier with a name that is the same as a reserved word.

The reserved keywords are:

AND	DAY_OF_WEEK	HAVING	OR	SYSDATE
AS	DAY_OF_YEAR	HOUR	ORDER	SYSTIMESTAMP
ASC	DEFINE	IN	PAGE	THEN
BETWEEN	DESC	IS	PERCENT	TRUE
BY	ELSE	JOIN	QUARTER	WEEK
CASE	END	JULIAN_DAY_NUMBER	RETURN	WHEN
CROSS	EVERY	LEFT	RIGHT	WHERE
CUBE	FALSE	MINUTE	ROLLUP	YEAR
CURRENT_DATE	FROM	MONTH	SECOND	
CURRENT_TIMESTAMP	FULL	NOT	SELECT	
DATE	GROUP	NULL	SETS	
DAY_OF_MONTH	GROUPING	ON	SOME	

In addition, MEMBERS and SATISFIES are keywords that are reserved for future use (and as such, are not documented in this guide).

Keep in mind that function names (such as COUNT and STRING_JOIN) are not keywords and, therefore, could be used as identifiers. However, as a best practice, you should also avoid using function names as identifiers.

Reserved punctuation symbols

- , (comma)
- ; (semi-colon)
- . (dot)
- / (division)
- + (plus)
- - (minus)

- * (star)
- < (less than)
- > (greater than)
- <= (less than or equal)
- => (greater than or equal)
- = (equal)
- <> (not equal)
- ((left parenthesis)
-) (right parenthesis)
- { (left brace)
- } (right brace)
- [(left bracket)
-] (right bracket)

Chapter 2

Statements and Clauses

This chapter describes the types of clauses used in EQL statements.

For information on the `GROUP` and `GROUP BY` clauses, see *Aggregation on page 27*.

DEFINE clause

`DEFINE` is used to generate an intermediate result that will not be included in the query result.

All EQL statements begin with either `DEFINE` or `RETURN`.

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.

The `DEFINE` syntax is:

```
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 clause

`RETURN` indicates that the statement result should be included in the query result.

All EQL statements begin with either `DEFINE` or `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.

The `RETURN` syntax is:

```
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 <attributeKey>[, <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.

Using SELECT *

`SELECT *` selects all the attributes at once from a given record source. The rules for using `SELECT *` are:

- You cannot use `SELECT *` over the corpus. This means that you must use a `FROM` clause in your statement to reference a non-corpus source.
- You cannot use `SELECT *` in a grouping statement.

For example, assume this simple query:

```
DEFINE ResellerInfo as
SELECT DimReseller_ResellerName, DimGeography_StateProvinceName, DimReseller_Phone;

RETURN Resellers as
SELECT *
FROM ResellerInfo
```

The query first generates an intermediate result (named ResellerInfo) from data in three attributes, and then uses `SELECT *` to select all the attributes from ResellerInfo.

You can also use `SELECT *` with a `JOIN` clause, as shown in this example:

```
DEFINE Reseller AS
SELECT DimReseller_ResellerKey, DimReseller_ResellerName, DimReseller_AnnualSales;

DEFINE Orders AS
SELECT FactSales_ResellerKey, FactSales_SalesAmount;

RETURN TopResellers AS
SELECT R.*, O.FactSales_SalesAmount
FROM Reseller R JOIN Orders O on (R.DimReseller_ResellerKey = O.FactSales_ResellerKey)
WHERE O.FactSales_SalesAmount > 10000
```

In the example, the expression `R.*` (in the `RETURN TopResellers` statement) expands to include all the attributes selected in the `DEFINE Reseller` statement.

Note you should be aware of the behavior of `SELECT *` clauses in regard to attributes with the same name in statements. That is, assuming these scenarios:

```
SELECT Amt AS Z, *
   or
SELECT *, Amt AS Z
```

if ***** includes an attribute named ***Z***, then whichever attribute comes first is the one included in the result.

Likewise in a join:

```
SELECT * FROM a JOIN b ON (...)
```

If a and b both contain an attribute with the same name, then you get the attribute from the first statement in the `JOIN` clause.

AS clause

The AS clause allows you to give an alias name to EQL attributes and results.

The alias name can be given to an attribute, attribute list, expression result, or query result set. The aliased name is temporary, as it does not persist across different EQL queries.

Alias names must be NCName-compliant (for example, they cannot contain spaces). 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/*.

Note: Attribute names are not required to be aliased, as the names are already NCName-compliant. However, you can alias attribute names if you wish (for example, for better human readability of a query that uses long attribute names).

AS is used in:

- `DEFINE` statements, to name a record set that will later be referenced by another statement (such as a `SELECT` or `FROM` clause).
- `RETURN` statements, to name the EQL results. This name is typically shown at the presentation level.
- `SELECT` statements, to name attributes, attribute lists, or expression results. This name is also typically shown at the presentation level.

Assume this `DEFINE` example:

```
DEFINE EmployeeTotals AS
SELECT
  DimEmployee_FullName AS Name,
  SUM(FactSales_SalesAmount) AS Total
```

```
GROUP BY DimEmployee_EmployeeKey, ProductSubcategoryName;
...
```

In the example, **EmployeeTotals** is an alias for the results produced by the `SELECT` and `GROUP BY` statements, while **Name** is an alias for the DimEmployee_FullName attribute and **Total** is an alias for the results of the `SUM` expression.

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
```

FROM clauses

You can include a `FROM` clause in your statement to specify a different record source from 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, based on a relationship between certain attributes in these statements.

`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 <Statement1> [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: If not used correctly, joins can cause the Endeca Server to grow beyond available RAM because they can easily create very large results. Two best practices are to avoid `CROSS JOIN` if possible and to be careful with `ON` conditions so that the number of results are reasonable.

WHERE clauses

The `WHERE` clause is used to filter input records for an expression.

EQL provides two filtering options: `WHERE` and `HAVING`. The syntax of the `WHERE` clause is as follows:

```
WHERE <BooleanExpression>
```

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 70*.

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 52*.

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).

HAVING clauses

The `HAVING` clause is used to filter output records.

EQL provides two filtering options: `WHERE` and `HAVING`.

The `HAVING` syntax is as follows:

```
HAVING <BooleanExpression>
```

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.

You can sort result records by specifying attribute names or an arbitrary expression.

The `ORDER BY` syntax is as follows:

```
ORDER BY <Attr/Exp> [ASC/DESC] [,<AttrExp> [ASC/DESC]]*
```

where *Attr/Exp* is either an attribute name or an arbitrary expression.

Optionally, you 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 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:

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

String sorting

String values are sorted in Unicode code point order.

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.

Expression sorting

An ORDER BY clause allows you to use an arbitrary expression to sort the resulting records. The expressions in the ORDER BY clause will only be able to refer to attributes of the local statement, except through LOOKUP expressions, as shown in these simple statements:

```
/* Invalid statement */
DEFINE T1 AS
SELECT ... AS foo

RETURN T2 AS
SELECT ... AS bar
FROM T1
ORDER BY T1.foo  /* not allowed */

/* Valid statement */
DEFINE T1 AS
SELECT ... AS foo

RETURN T2 AS
SELECT ... AS bar
FROM T1
ORDER BY T1[].foo  /* allowed */
```

In addition, the expression cannot contain aggregation functions. For example:

```
RETURN T AS
SELECT ... AS bar
FROM T1
ORDER BY SUM(bar) /* not allowed because of SUM aggregation function */

RETURN T AS
SELECT ... AS bar
FROM T1
ORDER BY ABS(bar) /* allowed */
```

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. The order can be impacted by changes such as 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.

Note that `ORDER BY` only impacts the result of a `RETURN` clause, or the effect of a `PAGE` clause. `ORDER BY` on a `DEFINE` with no `PAGE` clause has no effect.

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)

Chapter 3
Aggregation

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

GROUP/GROUP BY clauses

The `GROUP` and `GROUP BY` clauses specify how to map source records to result records in order to group statement output.

Some of the ways to use these clauses in a query are:

- 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 <attributeList>` maps source records to result records by the combination of values in the listed attributes.

You can also use other grouping functions (such as `CUBE` or `GROUPING SETS`) with the `GROUP` and `GROUP BY` clauses. Details on these functions are given in later in this chapter.

BNF grammar for grouping

The BNF grammar representation for `GROUP` and the family of group functions is:

```
GroupByList ::= GROUP | GROUP BY GroupByList_ | GROUP BY GroupAll
GroupByList_ ::= GroupByElement | GroupByList_ ,  GroupByElement
GroupByElement ::= GroupBySingle | GroupingSets | CubeRollup

GroupingSets ::= GROUPING SETS (GroupingSetList)
GroupingSetList ::=  GroupingSetElement |  GroupingSetList ,  GroupingSetElement
GroupingSetElement ::=  GroupBySingle | GroupByComposite | CubeRollup | GroupAll
```

```
CubeRollup ::= {CUBE | ROLLUP} (CubeRollupList)
CubeRollupList ::=  CubeRollupElement |  CubeRollupList ,  CubeRollupElement
CubeRollupElement ::=  GroupBySingle | GroupByComposite

GroupBySingle ::= Identifier
GroupByComposite ::= (GroupByCompositeList)
GroupByCompositeList ::= GroupBySingle | GroupByCompositeList, GroupBySingle

GroupAll ::= ()
```

Note that the use of GroupAll results in the following being all equivalent:

```
GROUP = GROUP BY() = GROUP BY GROUPING SETS(())
```

Specifying only GROUP

You can use a `GROUP` clause to aggregate results into a single bucket. As the BNF grammar shows, the `GROUP` clause does not take an argument.

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

```
RETURN ReviewCount AS
SELECT SUM(NumReviews) AS NumberOfReviews
GROUP
```

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

Specifying GROUP BY

You can use `GROUP BY` to aggregate results into buckets with common values for the grouping keys. The `GROUP BY` syntax is:

```
GROUP BY(attributeList)
```

where *attributeList* is a single attribute, a comma-separated list of multiple attributes, `GROUPING SETS`, `CUBE`, `ROLLUP`, or () to specify an empty group. The empty group generates a total.

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. However, `LOOKUP` and `IN` expressions are valid in this context.

All grouping attributes are part of the result records. A NULL value in any grouping attribute is treated like any other value, which means the source record is mapped to result records. (However, note that NULL values are ignored if selecting from the corpus.) For information about user-defined NULL-value handling in EQL, see *COALESCE on page 68*.

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" }
```

Note that using duplicated columns in `GROUP BY` clauses is allowed. This means that the following two queries are treated as equivalent:

```
RETURN Results AS
SELECT SUM(PROMO_COST) AS PR_Cost
GROUP BY PROMO_NAME

RETURN Results AS
SELECT SUM(PROMO_COST) AS PR_Cost
GROUP BY PROMO_NAME, PROMO_NAME
```

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. However, `GROUP BY` statements cannot use the `ANCESTOR` function (because you cannot group by an expression in EQL). Therefore, you must first use `ANCESTOR` with the `SELECT` statement and then specify the aliased results in the `GROUP BY` clause.

For example, assume that the Region attribute contains the hierarchy Country, State, and City. We want to group the results at the State level (one level below the root of the managed attribute hierarchy). An abbreviated query would look like this:

```
SELECT ANCESTOR("Region", 1) AS StateInfo
...
GROUP BY StateInfo
```

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.

GROUPING SETS expression

A `GROUPING SETS` expression allows you to selectively specify the set of groups that you want to create within a `GROUP BY` clause.

`GROUPING SETS` specifies multiple groupings of data in one query. Only the specified groups are aggregated, instead of the full set of aggregations that are generated by `CUBE` or `ROLLUP`. `GROUPING SETS` can contain a single element or a list of elements. `GROUPING SETS` can specify groupings equivalent to those returned by `ROLLUP` or `CUBE`. The *GroupingSetList* can contain `ROLLUP` or `CUBE`.

GROUPING SETS syntax

The `GROUPING SETS` syntax is:

```
GROUPING SETS(groupingSetList)
```

where *groupingSetList* is a single attribute, a comma-separated list of multiple attributes, `CUBE`, `ROLLUP`, or () to specify an empty group. The empty group generates a total. Note that nested grouping sets are not allowed.

For example:

```
GROUP BY GROUPING SETS(a, (b), (c, d), ())
```

Multiple grouping sets expressions can exist in the same query.

```
GROUP BY a, GROUPING SETS(b, c), GROUPING SETS((d, e))
```

is equivalent to:

```
GROUP BY GROUPING SETS((a, b, d, e),(a, c, d, e))
```

Keep in mind that the use of () to specify an empty group means that the following are all equivalent:

```
GROUP = GROUP BY() = GROUP BY GROUPING SETS(())
```

Note: Multiple grouping sets cannot be used on the corpus.

How duplicate attributes in a grouping set are handled

Specifying duplicate attributes in a given grouping set will not raise an error, but only one instance of the attribute will be used. For example, these two queries are equivalent:

```
SELECT SUM(PROD_NAME) AS Products GROUP BY PROD_LIST_PRICE, PROD_LIST_PRICE

SELECT SUM(PROD_NAME) AS Products GROUP BY PROD_LIST_PRICE
```

However, you can use duplicate attributes if they are in different grouping sets. In this `GROUPING SETS` example:

```
GROUP BY GROUPING SETS((COUNTRY_TOTAL), (COUNTRY_TOTAL))
```

two "COUNTRY_TOTAL" groups are generated.

However, this example:

```
GROUP BY GROUPING SETS((COUNTRY_TOTAL, COUNTRY_TOTAL))
```

will generate only one "COUNTRY_TOTAL" group because both attributes are in the same grouping set.

GROUPING SETS example

```
DEFINE ResellerSales AS
SELECT SUM(DimReseller_AnnualSales) AS TotalSales,
  DimReseller_ResellerName AS RepNames,
  DimReseller_OrderMonth AS OrderMonth
GROUP BY OrderMonth;

RETURN MonthlySales AS
SELECT AVG(TotalSales) AS AvgSalesPerRep
FROM ResellerSales
GROUP BY TotalSales, GROUPING SETS(RepNames), GROUPING SETS(OrderMonth)
```

ROLLUP extension

`ROLLUP` is an extension to `GROUP BY` that enables calculation of multiple levels of subtotals across a specified group of attributes. It also calculates a grand total.

`ROLLUP` (like `CUBE`) is syntactic sugar for `GROUPING SETS`:

```
ROLLUP(a, b, c) = GROUPING SETS((a,b,c), (a,b), (a), ())
```

The action of `ROLLUP` is that it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the `ROLLUP` clause. `ROLLUP` takes as its argument an ordered list of attributes and works as follows:

1. It calculates the standard aggregate values specified in the `GROUP BY` clause.
2. It creates progressively higher-level subtotals, moving from right to left through the list of attributes.
3. It creates a grand total.
4. Finally, `ROLLUP` creates subtotals at *n*+1 levels, where *n* is the number of attributes.

   For instance, if a query specifies `ROLLUP` on attributes of `time`, `region`, and `department` (*n*=3), the result set will include rows at four aggregation levels.

In summary, `ROLLUP` is intended for use in tasks involving subtotals.

ROLLUP syntax

`ROLLUP` appears in the `GROUP BY` clause, using this syntax:

```
GROUP BY ROLLUP(attributeList)
```

where *attributeList* is either a single attribute or a comma-separated list of multiple attributes.

Note: `ROLLUP` cannot be used on the corpus.

ROLLUP example

```
DEFINE Resellers AS SELECT
  DimReseller_AnnualSales AS Sales,
  DimGeography_CountryRegionName AS Countries,
  DimGeography_StateProvinceName AS States,
  DimReseller_OrderMonth AS OrderMonth
WHERE DimReseller_OrderMonth IS NOT NULL;

RETURN ResellerSales AS
SELECT SUM(Sales) AS TotalSales
FROM Resellers
GROUP BY ROLLUP(Countries, States, OrderMonth)
```

Partial ROLLUP

You can also roll up so that only some of the subtotals are included. This partial rollup uses this syntax:

```
GROUP BY expr1, ROLLUP(expr2, expr3)
```

In this case, the `GROUP BY` clause creates subtotals at (2+1=3) aggregation levels. That is, at level (expr1, expr2, expr3), (expr1, expr2), and (expr1).

Using the above example, the `GROUP BY` clause for partial `ROLLUP` would look like this:

```
DEFINE Resellers AS SELECT
  ...
RETURN ResellerSales AS
SELECT SUM(Sales) AS TotalSales
FROM Resellers
GROUP BY Countries, ROLLUP(States, OrderMonth)
```

CUBE extension

`CUBE` takes a specified set of attributes and creates subtotals for all of their possible combinations.

If *n* attributes are specified for a `CUBE`, there will be 2 to the *n* combinations of subtotals returned.

`CUBE` (like `ROLLUP`) is syntactic sugar for `GROUPING SETS`:

```
CUBE(a, b, c) = GROUPING SETS((a,b,c), (a,b), (a,c), (b,c), (a), (b), (c), ())
```

CUBE syntax

`CUBE` appears in the `GROUP BY` clause, using this syntax:

```
GROUP BY CUBE(attributeList)
```

where *attributeList* is either one attribute or a comma-separated list of multiple attributes.

Note: `CUBE` cannot be used on the corpus.

CUBE example

This example is very similar to the `ROLLUP` example, except that it uses `CUBE`:

```
DEFINE Resellers AS SELECT
  DimReseller_AnnualSales AS Sales,
  DimGeography_CountryRegionName AS Countries,
  DimGeography_StateProvinceName AS States,
  DimReseller_OrderMonth AS OrderMonth
WHERE DimReseller_OrderMonth IS NOT NULL;

RETURN ResellerSales AS
SELECT SUM(Sales) AS TotalSales
FROM Resellers
GROUP BY CUBE(Countries, States, OrderMonth)
```

Partial CUBE

Partial `CUBE` is similar to partial `ROLLUP` in that you can limit it to certain attributes and precede it with attributes outside the `CUBE` operator. In this case, subtotals of all possible combinations are limited to the attributes within the cube list (in parentheses), and they are combined with the preceding items in the `GROUP BY` list.

The syntax for partial `CUBE` is:

```
GROUP BY expr1, CUBE(expr2, expr3)
```

This syntax example calculates 2^2 (i.e., 4) subtotals:

- (expr1, expr2, expr3)
- (expr1, expr2)
- (expr1, expr3)
- (expr1)

Using the above example, the `GROUP BY` clause for partial `CUBE` would look like this:

```
DEFINE Resellers AS SELECT
  ...
RETURN ResellerSales AS
SELECT SUM(Sales) AS TotalSales
FROM Resellers
GROUP BY Countries, CUBE(States, OrderMonth)
```

Grouping sets helper functions

There are three helping functions that you can use for queries that use grouping capabilities.

`GROUPING`, `GROUPING_ID`, and `GROUP_ID` are helping functions for `GROUPING SETS`, `CUBE`, and `ROLLUP`. Note that these helping functions cannot be used in a `WHERE` clause, join condition, inside an aggregate function, or in the definition of a grouping attribute.

GROUPING function

GROUPING_ID function

GROUP_ID function

GROUPING function

`GROUPING` indicates whether a specified attribute expression in a `GROUP BY` list is aggregated.

The use of `ROLLUP` and `CUBE` can result in two challenging problems:

- How can you programmatically determine which result set rows are subtotals, and how do you find the exact level of aggregation for a given subtotal? You often need to use subtotals in calculations such as percent-of-totals, so you need an easy way to determine which rows are the subtotals.
- What happens if query results contain both stored NULL values and NULL values created by a `ROLLUP` or `CUBE`? How can you differentiate between the two?

The `GROUPING` function can handle these problems.

`GROUPING` is used to distinguish the NULL values that are returned by `ROLLUP`, `CUBE`, or `GROUPING SETS` from standard null values. The NULL returned as the result of a `ROLLUP`, `CUBE`, or `GROUPING SETS` operation is a special use of NULL. This acts as a column placeholder in the result set and means all.

`GROUPING` returns TRUE when it encounters a NULL value created by a `ROLLUP`, `CUBE`, or `GROUPING SETS` operation. That is, if the NULL indicates the row is a subtotal, `GROUPING` returns TRUE. Any other type of value, including a stored NULL, returns FALSE.

`GROUPING` thus lets you programmatically determine which result set rows are subtotals, and helps you find the exact level of aggregation for a given subtotal.

GROUPING syntax

The `GROUPING` syntax is:

```
GROUPING(attribute)
```

where *attribute* is a single attribute.

GROUPING example

```
DEFINE r AS SELECT
  DimReseller_AnnualRevenue AS Revenue,
  DimReseller_AnnualSales AS Sales,
  DimReseller_OrderMonth AS OrderMonth;

RETURN results AS SELECT
  COUNT(1) AS COUNT,
  GROUPING(Revenue) AS grouping_Revenue,
  GROUPING(Sales) AS grouping_Sales,
  GROUPING(OrderMonth) AS grouping_OrderMonth
FROM r
GROUP BY
  GROUPING SETS (
    ROLLUP(
      (Revenue),
      (Sales),
      (OrderMonth)
    )
  )
```

GROUPING_ID function

The `GROUPING_ID` function computes the `GROUP BY` level of a particular row.

The `GROUPING_ID` function returns a single number that enables you to determine the exact `GROUP BY` level. For each row, `GROUPING_ID` takes the set of 1's and 0's that would be generated if you used the appropriate `GROUPING` functions and concatenated them, forming a bit vector. The bit vector is treated as a binary number, and the number's base-10 value is returned by the `GROUPING_ID` function.

For example, if you group with the expression `CUBE(a, b)`, the possible values are:

Aggregation Level	Bit Vector	GROUPING_ID
a,b	0 0	0
a	0 1	1
b	1 0	2
Grand Total	1 1	3

GROUPING_ID syntax

The `GROUPING_ID` syntax is:

```
GROUPING_ID(attributeList)
```

where *attributeList* is a single attribute or a comma-separated list of 1-63 attributes.

GROUPING_ID example

```
DEFINE r AS SELECT
  DimReseller_AnnualRevenue AS Revenue,
  DimReseller_AnnualSales AS Sales;

RETURN results AS SELECT
  COUNT(1) AS COUNT,
  GROUPING_ID(Revenue) AS gid_Revenue,
  GROUPING_ID(Sales) AS gid_Sales
FROM r
GROUP BY CUBE(Revenue,Sales)
```

GROUP_ID function

The `GROUP_ID` function uniquely identifies a group of rows that has been created by a `GROUP BY` clause in the query result set.

The `GROUP BY` extensions (such as `CUBE`) allow complex result sets that can include duplicate groupings. The `GROUP_ID` function allows you to distinguish among duplicate groupings.

If there are multiple sets of rows calculated for a given level, `GROUP_ID` assigns the value of 0 to all the rows in the first set. All other sets of duplicate rows for a particular grouping are assigned higher values, starting with 1.

GROUP_ID thus helps you filter out duplicate groupings from the result set. For example, you can filter out duplicate groupings by adding a HAVING clause condition GROUP_ID()=0 to the query.

GROUP_ID syntax

GROUP_ID cannot be used in a WHERE clause, join condition, inside an aggregate function, or in the definition of a grouping attribute.

The GROUP_ID syntax is:

```
GROUP_ID() AS alias
```

Note that the function does not accept any parameters.

GROUP_ID example

```
DEFINE r AS SELECT
  DimReseller_AnnualRevenue AS Revenue,
  DimReseller_AnnualSales AS Sales,
  DimReseller_OrderMonth AS OrderMonth;

RETURN results AS SELECT
  COUNT(1) AS COUNT,
  GROUP_ID() AS gid,
  GROUPING(Revenue) AS grouping_Revenue,
  GROUPING(Sales) AS grouping_Sales,
  GROUPING(OrderMonth) AS grouping_OrderMonth
FROM r
GROUP BY OrderMonth, ROLLUP(Revenue,Sales)
```

Notes on grouping behavior

This topic describes some EQL grouping behaviors that you should be aware of.

Implicit nested aggregation

Nested aggregation is not allowed in EQL (that is, an aggregation function cannot be nested inside another aggregation function). The following are examples of explicit nested aggregation functions:

```
SELECT COUNT(SUM(sales)) AS totals GROUP // Invalid

SELECT COUNT(sales) AS totals, SUM(totals) AS totals2 GROUP // Invalid
```

However, EQL allows adding an implicit ARB. For example, the following two queries are equivalent:

```
SELECT sales AS totals GROUP // Valid implicit ARB

SELECT ARB(sales) AS sales GROUP // Valid explicit ARB
```

Therefore, the implicit ARB can result in implicit nested aggregation. For example, the following two queries are equivalent and will result in implicit nested aggregation:

```
SELECT sales AS totals, COUNT(totals) AS totals2 GROUP // Invalid

SELECT ARB(sales) AS totals, COUNT(totals) AS totals2 GROUP // Invalid
```

Note that setting the alias to be the same name as the selected attribute can make nested aggregation less obvious. The following two queries are equivalent and invalid:

```
SELECT sales AS sales, COUNT(sales) AS totals GROUP // Invalid

SELECT ARB(sales) AS sales, COUNT(sales) AS totals GROUP // Invalid
```

The solution is to use a different alias, as in these two queries, which are equivalent and valid:

```
SELECT sales AS sales1, COUNT(sales) AS totals GROUP // Valid

SELECT ARB(sales) AS sales1, COUNT(sales) AS totals GROUP // Valid
```

GROUPING and GROUPING_ID interaction with attribute source

Setting an alias to be the same as a selected attribute can change the attribute source. For example, in the following query, **amount** in **stmt1_amount** refers to **stmt1.amount**, while **amount** in **stmt2_amount** refers to **stmt2.amount**:

```
SELECT stmt1 AS SELECT amount AS amount;
SELECT stmt2 AS SELECT amount+1 AS stmt1_amount, amount+2 AS amount, amount
+3 AS stmt2_amount FROM stmt1
```

This also applies when using the `GROUPING` and `GROUPING_ID` functions:

```
SELECT stmt1 AS SELECT amount AS amount;
SELECT GROUPING(amount) AS stmt1_amount, amount AS amount,
  GROUPING(amount) AS stmt2_amount, orders AS orders,
  FROM stmt1
  GROUP BY CUBE(amount, orders)
```

Implicit selects

Implicit selects are added to the end of the `select` list. For example, the following two queries are equivalent:

```
SELECT COUNT(sales) AS cnt GROUP BY totals, price

SELECT COUNT(sales) AS cnt, totals AS totals, price AS price GROUP BY totals, price
```

This only affects constructs that have different pre-aggregate and post-aggregate behavior, such as the `GROUPING` function.

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 non-NULL 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.

Note that the `COUNT(1)` syntax returns all records, including those with NULL values. For example, you can get the number of records in your data store as follows:

```
RETURN results AS
SELECT COUNT(1) AS recordCount
GROUP
```

The statement result should be an integer that represents the total number of records in your data set.

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.

You should also take care when using `COUNTDISTINCT` on the corpus. For example, this statement:

```
SELECT COUNTDISTINCT(multiassign_attr) AS n FROM AllBaseRecords
```

will de-multiassign prior to aggregating, so it will not return the correct answer.

The correct way to do this is to group by the attribute, then count the results:

```
DEFINE a AS SELECT 1 AS n
FROM AllBaseRecords
GROUP BY multiassign_attr;

RETURN b AS
SELECT COUNT(n) AS n
FROM a
```

Multi-level 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
```

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)`)

Chapter 4

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.

EQL data type	Description	Related Dgraph data type
`string`	Represents character strings.	`mdex:string`
`int`	Represents a 64-bit integer. See note below on integers.	`mdex:long`
`double`	Represents a floating point number.	`mdex:double`
`boolean`	Represents a Boolean value (`TRUE` or `FALSE`)	`mdex:boolean`
`time`	Represents the time of day to a resolution of milliseconds.	`mdex:time`
`dateTime`	Represents a date and time to a resolution of milliseconds.	`mdex:dateTime`

EQL data type	Description	Related Dgraph data type
`duration`	Represents a length of time with a resolution of milliseconds.	`mdex:duration`
`geocode`	Represents a latitude and longitude pair.	`mdex:geocode`

Note on integer data type

While Dgraph records support both 32-bit integers (`mdex:int` data type) and 64-bit integers (`mdex:long` data type), EQL only supports 64-bit integers. This means that if you query an attribute that has a 32-bit value, it will appear as a long (64-bit value) in EQL results.

Operator precedence rules

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 that 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`

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

Literals

This section discusses how literals are used in EQL.

Character handling

Handling of upper- and lower-case

Handling NULL attribute values

Type promotion

Handling of NaN, inf, and -inf results

Character handling

EQL accepts all Unicode characters.

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

Literal type	Handling

String literals	String literals must be surrounded by single quotation marks.
	Embedded single quotes and backslashes must be escaped by backslashes. Examples:
	`'jim'`
	`'àlêx\'s house'`

Numeric literals	Numeric literals can be integers or floating point numbers.
	Numeric literals cannot be surrounded by single quotation marks.
	Numeric literals do not support exponential notation, and they cannot have trailing f|F|d|D to indicate float or double.
	`34`
	`.34`

Boolean literal	`TRUE/FALSE` 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:
	`RETURN Result AS`
	`SELECT  TO_GEOCODE(45.0, 37.0) AS Geocode,`
	`        TO_DATETIME('2012-11-21T08:22:00Z') AS Timestamp`

Literal type	Handling

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/*.
	An identifier must be enclosed in double quotation marks if:
	• The identifier contains characters other than letters, digits, and underscores.
	• The identifier starts with a digit.
	• The identifier uses the same name as an EQL reserved keyword. For example, if an attribute is named `WHERE` or `GROUP`, then it must be specified as `"WHERE"` or `"GROUP"`.
	If an identifier is in quotation marks, then you must use a backslash to escape double quotation marks and backslashes.
	Examples:
	`"Count"`
	`"Sales.Amount"`

Handling of upper- and lower-case

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. 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.
Aggregating functions	EQL ignores records with NULL values. 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. If all 10 records have a NULL Price, then most aggregations, such as `SUM(Price)`, also result in NULL values. 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). Note, however, that `COUNT(1)` does count records with NULL values.
Boolean operators	See *Boolean operators on page 65*.
Grouping expressions	If grouping from intermediate results, EQL does not ignore records that have a NULL value in any of the group keys, and considers the record to be present in a group. Even all-NULL groups are returned. If grouping from the corpus, EQL ignores records that have a NULL value in any of the group keys, and does not consider the record to be present in any group.

Type of operation	How EQL handles NULL values
Filters	When doing a comparison against a specific value, the NULL value will not match the specified filter, except for the `IS NULL` filter. For example, if record A has price 5, and record B has no price value, then: • `WHERE price = 5` matches A • `WHERE NOT(price = 5)` matches neither A nor B • `WHERE price <> 5` matches neither A nor B • `WHERE NOT(price <> 5)` matches A • `WHERE price = 99` matches neither A nor B • `WHERE NOT(price = 99)` matches A • `WHERE price <> 99` matches A • `WHERE NOT(price <> 99)` matches neither A nor B
Sorting	For any sort order specified, EQL returns: 1. Normal results 2. Records for a NaN value 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 general, EQL performs type promotion when there is no risk of loss of information.

In some cases, EQL supports automatic value promotion of integers (to doubles) and strings (to managed attribute values).

Promotion of integers

Promotion of integers to doubles occurs in the following contexts:

- Arguments to the `COALESCE` expression when called with a mix of integer and double.
- Arguments to the following operators when called with a mix of integer and double:

  `+ - * = <>`
- Integer arguments to the following functions are always converted to double.
  - / (division operator; note that `duration` arguments are not converted)
  - `CEIL`
  - `COS`

  - `EXP`
  - `FLOOR`
  - `LN`
  - `LOG`
  - `SIN`
  - `MOD`
  - `POWER`
  - `SIN`
  - `SQRT`
  - `TAN`
  - `TO_GEOCODE`
  - `TRUNC`
- When the clauses in a `CASE` expression return a mix of integer and double results, the integers are promoted to double.

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.

In contexts other than the above, 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 such as the following is required:

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

Promotion of strings

Strings can also be promoted to managed attribute values. These strings must be string literals; other kinds of expressions that produce strings are not converted.

String promotion applies to arguments to the following functions when they are called with a mix of string and managed-attribute arguments:

- `CASE`
- `COALESCE`
- `GET_LCA`
- `IS_ANCESTOR`
- `IS_DESCENDANT`

For example, in `CASE` expressions, if some clauses produce values in a managed attribute hierarchy and others produce string literals, then the string literals are automatically converted to values in the hierarchy.

Note that for all the functions listed above, all managed-attribute arguments must be from the same underlying hierarchy.

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 `!=`). Any other comparison involving a `NaN` value is false.
Sorting	For any sort order specified, EQL returns: 1. Normal records 2. Records with a `NaN` value 3. Records with a NULL value

The following example shows how `inf` and `-inf` values are treated in ascending and descending sort orders:

```
ASC         DESC
----        ----
-inf        +inf
-4          3
0           0
3           -4
+inf        -inf
NaN         NaN
NULL        NULL
```

Functions and operators

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

Numeric functions

Aggregation functions

Hierarchy functions

Geocode functions

Date and time functions

String functions

Arithmetic operators

Boolean operators

Numeric functions

EQL supports the following numeric functions.

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

Function	Description and Example
FLOOR	Returns the largest integer value not greater than `n`. `SELECT FLOOR(123.45 AS x, FLOOR(32) AS y, FLOOR(-123.45) AS z` **RESULT:** `x` = `123, y` = `32, z` = `124`
LN	Natural logarithm. Computes the logarithm of its single argument, the base of which is `e`. `SELECT LN(1.0) AS baseE` **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. `Log(10,1000) = 3`
MOD	Modulo. Returns the remainder of `n` divided by `m`. `Mod(10,3) = 1` EQL uses the `fmod` floating point remainder, as defined in the C/POSIX standard.

ROUND	Returns a number rounded to the specified decimal place.
	The unary version takes only one argument (the number to be rounded) and drops the decimal (non-integral) portion of the input. For example:
	`ROUND(8.2) returns 8`
	`ROUND(8.7) returns 9`
	The binary version takes two arguments (the number to be rounded and a positive or negative integer that allows you to set the number of spaces at which the number is rounded):
	• Positive second arguments correspond to the number of places that must be returned ***after*** the decimal point. For example:
	`ROUND(123.4567, 3) = 123.457`
	• Negative second arguments correspond to the number of places that must be returned ***before*** the decimal point. For example:
	`ROUND(123.4567, -3) = 100.0`

SIGN	Returns the sign of the argument as -1, 0, or 1, depending on whether `n` is negative, zero, or positive. `SELECT SIGN(-12) AS x, SIGN(0) AS y, SIGN(12) AS z` **RESULT:** `x` = `-1, y` = `0, z` = `1`

Function	Description and Example
SQRT	Returns the nonnegative square root of `n`. `SELECT SQRT(9) AS x` **RESULT:** `x = 3`
TRUNC	Returns the number `n`, truncated to `m` decimal places. If `m` is 0, the result has no decimal point or fractional part. The unary version drops the decimal (non-integral) portion of the input, while the binary version allows you to set the number of spaces at which the number is truncated. `SELECT TRUNC(3.14159265, 3)as x` **RESULT:** `x = 3.141`
SIN	The sine of `n`, where the angle of `n` is in radians. `SIN(Pi/6) = 5`
COS	The cosine of `n`, where the angle of `n` is in radians. `COS(Pi/3) = .5`
TAN	The tangent of `n`, where the angle of `n` is in radians. `TAN(Pi/4) = 1`
POWER	Returns the value of `n` raised to the power of `m`. `Power(2,8) = 256`
TO_DURATION	Casts a string representation of a timestamp into a number of milliseconds so that it can be used as a duration.
TO_DOUBLE	Casts a string representation of an integer as a double.
TO_INTEGER(Boolean)	Casts `TRUE/FALSE` to `1/0`.

Aggregation functions

EQL supports the following aggregation functions.

Function	Description
ARB	Selects an arbitrary but consistent value from the set of values in a field.
AVG	Computes the arithmetic mean value for a field.

Function	Description
COUNT	Counts the number of records with valid non-NULL values in a field for each `GROUP BY` result.
COUNTDISTINCT	Counts the number of unique, valid non-NULL values in a field for each `GROUP BY` result.
EVERY	In a Boolean expression, returns `TRUE` if all values in a field are `TRUE`, otherwise returns `FALSE`.
MAX	Finds the maximum value for a field.
MIN	Finds the minimum value for a field.

MEDIAN	Finds the median value for a field (Note that `PAGE PERCENT` provides overlapping functionality).
	Note that the EQL definition of `MEDIAN` is the same as 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:
	`RETURN results AS SELECT`
	`  MEDIAN(a) AS med`
	`GROUP`
	produces the mean of the two elements in the middle of the sorted set, or 2.5.

SOME	In a Boolean expression, returns `TRUE` if at least one value in a field is `TRUE`, otherwise returns `FALSE`.
STDDEV	Computes the standard deviation for a field.
STRING_JOIN	Creates a single string containing all the values of a string attribute.
SUM	Computes the sum of field values.
VARIANCE	Computes the variance (that is, the square of the standard deviation) for a field.

STRING_JOIN function

The `STRING_JOIN` function takes a string property and a delimiter and creates a single string containing all of the property's values, separated by the delimiter. Its syntax is:

```
STRING_JOIN('delimiter', string_attribute)
```

The delimiter is a string literal enclosed in single quotation marks.

The resulting strings are sorted in an arbitrary but stable order within each group. NULL values are ignored in the output, but values having the empty string are not.

For this sample query, assume that the `R_NAME` standard attribute is of type string and contains names of regions, while the `R_NAME` standard attribute is also of type string and contains the names of nations:

```
RETURN results AS SELECT
  STRING_JOIN(',',R_NAME) AS Regions,
  STRING_JOIN(',',N_NAME) AS Nations
GROUP
```

The query will return the region and country names delimited by commas:

```
Nations
ALGERIA, ARGENTINA, BRAZIL, CANADA, CHINA, EGYPT, ETHIOPIA, FRANCE, GERMANY, INDIA, INDONESIA, IRAN,
IRAQ, JAPAN, JORDAN, KENYA, MOROCCO, MOZAMBIQUE, PERU, ROMANIA, RUSSIA, SAUDI ARABIA, UNITED KINGDOM,
UNITED STATES, VIETNAM
Regions
AFRICA,AMERICA,ASIA,EUROPE,MIDDLE EAST
```

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.
`TO_MANAGED_VALUE(attribute, value)`	Returns a managed value literal from literals representing a managed attribute and a managed value. Both parameters must be string literals.
`IS_DESCENDANT(attribute, value)`	Include the record if the named attribute is the attribute specified or a descendant and if the specified value matches. 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. This function can also be used with standard attributes. In this case, the record is included if the specified attribute exists and the specified value matches.
`IS_ANCESTOR(attribute, value)`	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. This function can also be used with standard attributes. In this case, the record is included if the specified attribute exists and the specified value matches.

Function	Description
`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.

Hierarchy examples

Example 1: In this 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')
```

Example 2: In this 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
```

Example 3: 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 cannot use the `ANCESTOR` function, because you cannot group by an expression in EQL.

Example 4: This abbreviated example shows the use of the `TO_MANAGED_VALUE` function:

```
RETURN Results AS
SELECT
  HIERARCHY_LEVEL(TO_MANAGED_VALUE('ProductCategory', 'Bikes')) AS HL
...
```

Geocode functions

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

Note that all distances are expressed in kilometers.

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:double, mdex:double)`	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.

Date and time functions

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 a time zone to 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` `SYSTIMESTAMP`	`dateTime` `dateTime`	Constants representing the current date and time (at an arbitrary point during query evaluation) in GMT and server time zone, respectively.
`CURRENT_DATE` `SYSDATE`	`dateTime` `dateTime`	Constants representing current date (at an arbitrary point during query evaluation) in GMT and server time zone, respectively.

`TO_TIME`
`TO_DATETIME`
`TO_DURATION`
	`time`
	`dateTime`
	`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` `FROM_TZ`	`dateTime` `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.

Function	Description	Format
`TO_TIME`	Constructs a timestamp representing time.	`<TimeStringFormat> ::= hh:mm:ss[.sss]((+|-) hh:mm |Z)`
`TO_DATETIME`	Constructs a timestamp representing date and time.	See the section below for the syntax of this function's string interface, date-only numeric interface, and date-time numeric interface.

Function	Description	Format

`TO_DURATION`	Constructs a timestamp representing duration.	`<DurationStringFormat> ::=`
		`[-]P[<Days>][T(<Hours>[<Minutes>}[<Seconds>]|`
		`<Minutes>[<Seconds>]|`
		`<Seconds>)]`
		`<Days> ::= <Integer>D`
		`<Hours> ::= <Integer>H`
		`<Minutes> ::= <Integer>M`
		`<Seconds> ::= <Integer>[.<Integer>]S`

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 time zone.

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

TO_DATETIME formats

The single-argument string interface for this function is:

```
TO_DATETIME(<DateTimeString>)
```

where:

```
<DateTimeString> ::= [-]YYYY-MM-DDT<TimeStringFormat>
```

Three examples of the string interface are listed in the table above.

The numeric interface signatures are:

```
TO_DATETIME(<Year>, <Month>, <Day>)

TO_DATETIME(<Year>, <Month>, <Day>, <Hour>, <Minute>, <Second>, <Millisecond>)
```

where all arguments are integers.

In the first signature, time arguments will be filled with zeros. In both signatures, time zone will be assumed to be UTC. If time zone information exists, duration (`TO_DURATION`) and time zone (`TO_TZ`) constructs can be used, as shown below in the examples.

Examples of the numeric interface signatures are:

```
TO_DATETIME(2012, 9, 22)

TO_DATETIME(2012, 9, 22, 23, 15, 50, 500)

TO_DATETIME(2012, 9, 22, 23, 15, 50, 500) + TO_DURATION(1000)

TO_TZ(TO_DATETIME(2012, 9, 22, 23, 15, 50, 500), 'America/New_York')
```

Time zone manipulation

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

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 time zone at that time in GMT, and returns a timestamp adjusted by that offset. If the specified time zone 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 time zone, looks up the GMT offset for the specified time zone at that time, and returns a timestamp adjusted by that offset. If the specified time zone 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 time zone 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 - 366)	
`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.
`YEAR`	(-9999 - 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

Expression	Results
`P5DT12H00M0.000S` / `P1DT0H00M0.000S`	5.5

String functions

EQL supports the following string functions.

Function	Description
`CONCAT`	Concatenates two string arguments into a single string.
`SUBSTR`	Returns a part (substring) of a character expression.
`TO_STRING`	Converts a value to a string.

CONCAT function

`CONCAT` is a row function that takes two string arguments and concatenates them into a single string. Its syntax is:

```
CONCAT(string1, string2)
```

Each argument can be a literal string (within single quotation marks), an attribute of type string, or any expression that produces a string.

This sample query uses literal strings for the arguments:

```
RETURN results AS SELECT
  CONCAT('Jane ', 'Wilson') AS FullName
GROUP
```

This similar query uses two string-type standard attributes:

```
RETURN results AS SELECT
  CONCAT(S_NAME,S_ADDRESS) AS Supplier
GROUP
```

SUBSTR function

The `SUBSTR` function has two syntaxes:

```
SUBSTR(string, position)

SUBSTR(string, position, length)
```

where:

- *string* is the string to be parsed.
- *position* is a number that indicates where the substring starts (counting from the left side). Note that the parameter is not zero indexed, which means that in order to start with the fifth character, the parameter has to be 5. If 0 (zero) is specified, it is treated as 1.
- *length* is a number that specifies the length of the substring that is to be extracted.

TO_STRING function

The `TO_STRING` function takes any data type value and returns a string equivalent. Its syntax is:

```
TO_STRING(input_value)
```

If the input value is NULL, the output value will also be NULL.

This sample query converts the value of the P_SIZE integer attribute to a string equivalent:

```
RETURN results AS SELECT
  TO_STRING(P_SIZE) AS Sizes
GROUP
```

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 43*.

Boolean operators

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

The results of Boolean operations (including the presence of NULL) is follows:

Value of a	Value of b	Result of a AND b	Result of a OR b	Result of NOT a
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

For information on order of operations, see *Operator precedence rules on page 43*.

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 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 subtotal 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
```

BETWEEN

The `BETWEEN` expression determines whether an attribute's value falls within a range of values.

`BETWEEN` is useful in conjunction with `WHERE` clauses.

The syntax for `BETWEEN` is:

```
<attribute> BETWEEN <startValue> AND <endValue>
```

where <*attribute*> is the attribute whose value will be tested. `BETWEEN` is inclusive, which means that it returns TRUE if the value of <*attribute*> is greater than or equal to the value of <*startValue*> and less than or equal to the value of <*endValue*>.

With one exception, <*attribute*> must be of the same data type as <*startValue*> and <*endValue*> (supported data types are integer, double, dateTime, duration, time, string, and Boolean). The exception is that you can use a mix of integer and double, because the integer is promoted to a double.

Note that if any of the `BETWEEN` arguments (i.e., <*attribute*>, <*startValue*>, or <*endValue*>) are NaN (Not a Number) values, then the expression evaluates to FALSE.

The following is a simple example of `BETWEEN`:

```
RETURN Results AS
SELECT SUM(AMOUNT_SOLD) AS SalesTotal
WHERE AMOUNT_SOLD BETWEEN 10 AND 100
GROUP BY CUST_STATE_PROVINCE
```

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
```

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. The result is `NULL` if the expression list does not match target group key values, or the target column is `NULL` for a matching target group key values.

Lookup attributes refer to `GROUP BY` clauses 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 5

EQL Use Cases

This chapter describes how to handle various business scenarios using EQL. The examples in this chapter 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
...
```

The query is:

```
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

ID	X	Y
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"
```

In the example, the words "one" and "many" are statement aliases to clarify the roles in this many-to-one self-join. Looking at the join condition, you can think of this as, for each (one) record, create multiple records based on the (many) values that match the join condition.

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 6

EQL Best Practices

This chapter 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.

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

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 this second example.

```
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 into a separate `SELECT`, then reuse it.

Index