Copyright and disclaimer

Copyright © 2003, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. UNIX is a registered trademark of The Open Group.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.
# Table of Contents

Copyright and disclaimer .......................................................... 2

Preface ........................................................................ 6
  About this guide ............................................................. 6
  Who should use this guide .................................................. 6
  Conventions used in this document ........................................ 6
  Contacting Oracle Customer Support ...................................... 7

Chapter 1: Introduction to the Endeca Query Language .................. 8
  EQL overview ............................................................... 8
  Important concepts and terms ............................................ 8
  EQL and SQL: a comparison ............................................... 9
  Query overview ............................................................ 10
  How queries are processed ............................................... 11
  EQL reserved keywords .................................................. 12

Chapter 2: Statements and Clauses ............................................ 14
  DEFINE clause ............................................................ 14
  RETURN clause ............................................................ 15
  LET clause ................................................................. 15
  SELECT clause ............................................................ 17
  AS clause ................................................................. 19
  FROM clause .............................................................. 20
  JOIN clause ............................................................... 21
  WHERE clause ............................................................ 25
  HAVING clause ........................................................... 26
  ORDER BY clause ......................................................... 27
  PAGE clause ............................................................... 30

Chapter 3: Aggregation .......................................................... 32
  GROUP/GROUP BY clauses ............................................. 32
  MEMBERS extension .................................................... 34
  GROUPING SETS expression .......................................... 37
  ROLLUP extension ....................................................... 38
  CUBE extension .......................................................... 39
  GROUPING function ..................................................... 40
  COUNT function ........................................................ 41
  COUNT_APPROX .......................................................... 43
  COUNTDISTINCT function ............................................. 43
  APPROXCOUNTDISTINCT function ................................. 44
  Multi-level aggregation .................................................. 45
  Per-aggregation filters .................................................... 46
Chapter 4: Expressions

Supported data types ................................................................. 47
Operator precedence rules .......................................................... 49
Literals ................................................................................ 50
  Character handling ............................................................... 50
  Handling of upper- and lower-case ........................................ 51
  Handling NULL attribute values .......................................... 52
  Integer type promotion ....................................................... 53
  Handling of NaN, inf, and -inf results ................................. 54

Functions and operators ............................................................ 55
  Numeric functions ................................................................ 56
  Aggregation functions .......................................................... 59
  Geocode functions ............................................................... 61
  Date and time functions ....................................................... 61
    Manipulating current date and time .................................... 63
    Constructing date and time values ................................... 63
    Time zone manipulation .................................................. 65
    Using EXTRACT to extract a portion of a dateTime value ..... 66
    Using TRUNC to round down dateTime values .................. 67
    Using arithmetic operations on date and time values ...... 68
  String functions .................................................................. 69
  Arithmetic operators ........................................................... 70
  Boolean operators ............................................................... 70

Using EQL results to compose follow-on queries ....................... 72
Using LOOKUP expressions for inter-statement references ....... 72

ARB .................................................................................. 74
BETWEEN ........................................................................ 76
CASE .............................................................................. 76
COALESCE ................................................................. 77
CORRELATION ............................................................ 78
HAS_REFINEMENTS ...................................................... 78
IN ............................................................................... 79
PERCENTILE .............................................................. 80
RECORD_IN_FAST_SAMPLE ............................................. 81

Chapter 5: Sets and Multi-assign Data ..................................... 83

About sets ........................................................................... 83
Aggregate functions .............................................................. 85
  SET function ................................................................ 85
  SET_INTERSECTIONS function ...................................... 87
  SET_UNIONS function .................................................. 89
Row functions ..................................................................... 90
  ADD_ELEMENT function ................................................ 90
  CARDINALITY function ................................................ 91
  DIFFERENCE function .................................................. 92
  FOREACH function ...................................................... 93
<table>
<thead>
<tr>
<th><strong>Table of Contents</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERSECTION function</strong></td>
<td>97</td>
</tr>
<tr>
<td><strong>IS_EMPTY and IS_NOT_EMPTY functions</strong></td>
<td>98</td>
</tr>
<tr>
<td><strong>IS_MEMBER_OF function</strong></td>
<td>100</td>
</tr>
<tr>
<td><strong>SINGLETON function</strong></td>
<td>102</td>
</tr>
<tr>
<td><strong>SUBSET function</strong></td>
<td>102</td>
</tr>
<tr>
<td><strong>TRUNCATE_SET function</strong></td>
<td>103</td>
</tr>
<tr>
<td><strong>UNION function</strong></td>
<td>104</td>
</tr>
<tr>
<td>Set constructor</td>
<td>105</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>106</td>
</tr>
<tr>
<td>Grouping by sets</td>
<td>109</td>
</tr>
<tr>
<td><strong>Chapter 6: EQL Use Cases</strong></td>
<td>111</td>
</tr>
<tr>
<td>Re-normalization</td>
<td>111</td>
</tr>
<tr>
<td>Grouping by range buckets</td>
<td>112</td>
</tr>
<tr>
<td>Manipulating records in a dynamically computed range value</td>
<td>113</td>
</tr>
<tr>
<td>Grouping data into quartiles</td>
<td>113</td>
</tr>
<tr>
<td>Combining multiple sparse fields into one</td>
<td>115</td>
</tr>
<tr>
<td>Joining data from different types of records</td>
<td>115</td>
</tr>
<tr>
<td>Linear regressions in EQL</td>
<td>116</td>
</tr>
<tr>
<td>Using an IN filter for pie chart segmentation</td>
<td>117</td>
</tr>
<tr>
<td>Running sum</td>
<td>117</td>
</tr>
<tr>
<td>Query by age</td>
<td>118</td>
</tr>
<tr>
<td>Calculating percent change between most recent month and previous month</td>
<td>118</td>
</tr>
<tr>
<td><strong>Chapter 7: EQL Best Practices</strong></td>
<td>120</td>
</tr>
<tr>
<td>Controlling input size</td>
<td>120</td>
</tr>
<tr>
<td>Filtering as early as possible</td>
<td>121</td>
</tr>
<tr>
<td>Controlling join size</td>
<td>122</td>
</tr>
<tr>
<td>Additional tips</td>
<td>122</td>
</tr>
</tbody>
</table>
Preface

Oracle Big Data Discovery is a set of end-to-end visual analytic capabilities that leverage the power of Hadoop to transform raw data into business insight in minutes, without the need to learn complex products or rely only on highly skilled resources.

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 document

The following conventions are used in this document.

Typographic conventions

The following table describes the typographic conventions used in this document.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Interface Elements</td>
<td>This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields.</td>
</tr>
<tr>
<td>Code Sample</td>
<td>This formatting is used for sample code phrases within a paragraph.</td>
</tr>
<tr>
<td>Variable</td>
<td>This formatting is used for variable values.</td>
</tr>
<tr>
<td></td>
<td>For variables within a code sample, the formatting is <strong>Variable</strong>.</td>
</tr>
<tr>
<td>File Path</td>
<td>This formatting is used for file names and paths.</td>
</tr>
</tbody>
</table>

Symbol conventions

The following table describes symbol conventions used in this document.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>The right angle bracket, or greater-than sign, indicates menu item selections in a graphic user interface.</td>
<td>File &gt; New &gt; Project</td>
<td>From the File menu, choose New, then from the New submenu, choose Project.</td>
</tr>
</tbody>
</table>
Path variable conventions

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

<table>
<thead>
<tr>
<th>Path variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MW_HOME</td>
<td>Indicates the absolute path to your Oracle Middleware home directory, which is the root directory for your WebLogic installation.</td>
</tr>
<tr>
<td>$DOMAIN_HOME</td>
<td>Indicates the absolute path to your WebLogic domain home directory. For example, if bdd_domain is the domain name, then the $DOMAIN_HOME value is the $MW_HOME/user_projects/domains/bdd_domain directory.</td>
</tr>
<tr>
<td>$BDD_HOME</td>
<td>Indicates the absolute path to your Oracle Big Data Discovery home directory. For example, if BDD1.0 is the name you specified for the Oracle Big Data Discovery installation, then the $BDD_HOME value is the $MW_HOME/BDD1.0 directory.</td>
</tr>
<tr>
<td>$DGRAPH_HOME</td>
<td>Indicates the absolute path to your Dgraph home directory. For example, the $DGRAPH_HOME value might be the $BDD_HOME/dgraph directory.</td>
</tr>
</tbody>
</table>

Contacting Oracle Customer Support

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

Chapter 1
Introduction to the Endeca Query Language

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

EQL overview

Important concepts and terms

EQL and SQL: a comparison

Query overview

How queries are processed

EQL reserved keywords

EQL overview

EQL is a SQL-like language designed specifically to query and manipulate data from the Dgraph.

EQL enables Dgraph–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 Dgraph. From the point of view of assignments on records, an attribute can be either:
  • **Single-assign attribute**: An attribute for which a record may have only one value. For example, because a book has only one price, the Price attribute would be single-assign. Single-assign attributes are of the atomic data type (such as `mdex:string` and `mdex:double`).
  • **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. Multi-assign attributes are of the set data type (such as `mdex:string-set` and `mdex:double-set`). They are represented in EQL by sets (see Sets and Multi-assign Data on page 82).

• **Record**: The fundamental unit of data in the Dgraph. 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. Records in collections can include multiple assignments to the same attribute, as can records in EQL results.
• **Collection**: The full body of Dgraph application records is contained in one or more collections (called *data sets* in Studio). Thus, Dgraph data is collection-based rather than table-based. By using a `FROM` clause in your statement, you can specify a named state (which in turn references a collection) to serve as the record source for your query. Alternatively, the `FROM` clause can specify a previously-defined statement as the record source. Note that a `FROM` clause is mandatory in an EQL statement.

• **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 Dgraph Gateway (and ultimately to the Dgraph). 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 collections 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 one or more collections 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. With the use of the `MEMBERS` extension in a `GROUP BY` clause, a single record can 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 comparisons:**
- `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 with at least one `RETURN` clause. 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 three clauses: a `DEFINE` or a `RETURN` clause, a `SELECT` clause, and a `FROM` clause. In addition, the statement 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:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square brackets [ ]</td>
<td>Optional</td>
<td><code>FROM &lt;statementKey&gt; [alias]</code></td>
</tr>
<tr>
<td>Asterisk *</td>
<td>May be repeated</td>
<td><code>{, JOIN statement [alias] ON &lt;Boolean expression&gt;}</code>*</td>
</tr>
<tr>
<td>Ellipsis ...</td>
<td>Additional, unspecified content</td>
<td><code>DEFINE &lt;recordSetName&gt; AS ...</code></td>
</tr>
</tbody>
</table>
### Commenting in EQL

You can comment your EQL code using the following notation:

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

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

```sql
RETURN Top5 AS SELECT SUM(Sale) AS Sales FROM SaleState GROUP BY Customer ORDER BY Sales DESC PAGE(0,5);
/*
RETURN Others AS SELECT SUM(Sale) AS Sales FROM SaleState 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 from a collection, the source records are the result records of the statement or the appropriately filtered collection 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 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, AS, ASC, BETWEEN, BY, CASE, COUNT, CROSS, CUBE, CURRENT, CURRENT_DATE,
CURRENT_TIMESTAMP, DATE, DAY_OF_MONTH, DAY_OF_WEEK, DAY_OF_YEAR, DEFINE, DESC, ELSE,
EMPTY, END, EVERY, FALSE, FOLLOWING, FOREACH, FROM, FULL, GROUP, GROUPING, HAVING,
HOUR, IN, INNER, IS, JOIN, JULIAN_DAY_NUMBER, LEFT, LET, MEMBERS, MINUTE, MONTH, NOT,
NULL, ON, OR, ORDER, OVER, PAGE, PARTITION, PERCENT, PRECEDING, QUARTER, RANGE, RETURN,
RIGHT, ROLLUP, SATISFIES, SECOND, SELECT, SETS, SOME, SYSDATE, SYSTIMESTAMP, THEN,
TRUE, UNBOUNDED, UNPAGED, VALUE, WEEK, WHEN, WHERE, WITH, YEAR
```

Keep in mind that many function names (such as SUM 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)
- ; (semicolon)
- . (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 section describes the types of clauses used in EQL statements.

For information on the GROUP and GROUP BY clauses, see Aggregation on page 31.

DEFINE clause
RETURN clause
LET clause
SELECT clause
AS clause
FROM clause
JOIN clause
WHERE clause
HAVING clause
ORDER BY clause
PAGE clause

**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 ...  
```

Note that the statement name cannot be the same as the state name or as any other statement.

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

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

RETURN ProductPct AS
SELECT 100*SUM(Amount) / RegionTotals[Region].Total AS PctTotal
FROM RegionTotals
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 Dgraph query result. This is important when more than one statement is submitted with the query.

The RETURN syntax is:

```
RETURN <statementName> AS ...
```

Note that the statement name cannot be the same as the state name or as any other statement.

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

```
RETURN Result AS 
SELECT COUNTDISTINCT(Color) AS Total 
FROM ProductState 
GROUP BY Size
```

WITH UNPAGED COUNT modifier

A RETURN clause can include an optional WITH UNPAGED COUNT modifier that computes the unpaged (total) record count for the statement and returns the count as in a NumRecords element in the results metadata. The syntax is:

```
RETURN <statementName> WITH UNPAGED COUNT AS ...
```

Assume, for example, this query:

```
RETURN Results WITH UNPAGED COUNT AS 
SELECT 
    WineType AS types, 
    Flavors AS tastes 
FROM winestate
```

If 50 records are returned, the metadata in the results would include this element:

```
NumRecords="50"
```

It would then be the responsibility of the application to parse this element and print number for the application’s UI. Note that NumRecords will still be 50 if you add, say PAGE(0, 10) to the statement.

Note the following about this parameter:

- WITH UNPAGED COUNT can be used in a statement with a PAGE clause.
- WITH UNPAGED COUNT is ignored if used in a DEFINE statement.

LET clause

The LET clause defines attributes that may be used elsewhere in the statement but do not necessarily appear in the statement’s result.

The primary intent of LET is to make it easier to group by the value of a computed attribute (and especially to group by the MEMBERS of a computed attribute). However, LET can be used in any statement, grouping or not, to define temporary values of use elsewhere in the statement.

The syntax is:
LET <expression> [AS <attribute>] [, <expression> [AS <attribute>] ]*

LET may appear in any statement, immediately before the SELECT clause, as in this example:

```
RETURN Results AS
LET
  x + y AS intermediateSum
SELECT
  MIN(x) AS min_x,
  intermediateSum + z AS finalSum
FROM WineState
GROUP BY finalSum
```

The output of the Results statement contains only two attributes, min_x and finalSum. The LET-bound attribute intermediateSum does not appear in the output.

If present, LET must appear immediately before the SELECT clauses (as in the example above) and it must be followed by one or more attribute definitions, separated by commas. These attribute definitions look and act exactly like those that appear after SELECT. In particular, if the expression on the left-hand side of the definition is a bare attribute reference (optionally with data-source qualifier), then the AS clause of the definition is optional. That is, you may write:

```
LET x, State.y AS y, 3 as z
```

which is equivalent to:

```
LET x AS x, State.y AS y, 3 as z
```

LET attributes are computed immediately after the statement WHERE clause (if used) and before any of the SELECT attributes are computed.

Because LET attributes are computed before grouping, aggregators like AVG and SUM are illegal in LET clauses, and EQL signals an error if any appear in that context.

**LET scoping**

An attribute defined with LET is in scope:

- for all following LET definitions in the same statement
- for all SELECT definitions in the same statement
- for the GROUP BY clause, including MEMBERS, in the same statement.

In addition, if a LET attribute is used as a grouping key, then it also appears in the statement's results and is available for use in ORDER BY and HAVING clauses. If the attribute is not a grouping key, then it is not in scope for ORDER BY or HAVING.

This example illustrates the LET scoping rules:

```
RETURN results AS
LET
  (FOREACH d IN orderDates RETURN (EXTRACT(d, YEAR))) AS orderYears
SELECT
  MAX(totalCost) AS maxCost
FROM OrderHistory
GROUP BY orderYears
HAVING 2014 IN orderYears
```
The example assumes that the data-source **OrderHistory** defines an attribute **orderDates** of type **mdex:dateTime-set**. The definition of the **orderYears** attribute extracts the year from each date in **orderDates** and then re-assembles these years into a set. The statement groups its results by the set of order years, computing the maximum cost for each, and returning rows for those orders that have at least one date in 2014 in **orderDates**. Because **orderYears** is a group key, the output table has two attributes (**maxCost** and **orderYears**), and **orderYears** is available for use in the **HAVING** clause.

As an alternative, consider this example:

```pc9
RETURN results AS
LET
  (FOREACH d IN orderDates RETURN (EXTRACT(d, YEAR))) AS orderYears
SELECT
  MAX(totalCost) AS maxCost
FROM OrderHistory
GROUP BY MEMBERS(orderYears) AS yr
```

This statement is the same as the previous example, except that this statement groups not by **orderYears** but rather by its members. Therefore, **orderYears** is not a group key, but is merely used to compute the group key **yr**. Therefore, **orderYears** does not appear in the statement's output, and it cannot appear in either **HAVING** or **ORDER BY** clauses. (The statement's output contains two attributes, **yr** and **maxCost**.)

To summarize the rules given above: in a non-grouping statement, **LET** attributes never appear in the output, and they are never visible in **HAVING** and **ORDER BY** clauses.

### SELECT clause

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

Its syntax is as follows:

```pc9
SELECT <expression> AS <attributeKey>{, <expression> AS <key>}
```

For example:

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

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

```sql
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 can use **SELECT** * over a collection. The statement's **FROM** clause specifies a named state (which in turn references a collection name). Keep in mind that retrieving all records from a very large collection can take some time.

- You cannot use the **AS** clause with a **SELECT** * statement. For example, this returns an error:

```sql
SELECT * AS allRecs
```

- You cannot use **SELECT** * in a grouping statement.
• **SELECT *** expansion will include grouping keys that are defined by a **LET** clause in the source statement.

For example, assume this simple query:

```plaintext
DEFINE ResellerInfo AS
SELECT
  DimReseller_ResellerName,
  DimGeography_StateProvinceName,
  DimReseller_Phone
FROM SaleState;

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.

The sample query selects all the attributes from a given collection:

```plaintext
RETURN Results as
SELECT *
FROM WineState
```

In the query, the WineState state references the Wines collection, which means that all of that collection’s records are returned.

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

```plaintext
DEFINE Reseller AS
SELECT
  DimReseller_ResellerKey,
  DimReseller_ResellerName,
  DimReseller_AnnualSales
FROM SaleState;

DEFINE Orders AS
SELECT
  FactSales_ResellerKey,
  FactSales_SalesAmount
FROM SaleState;

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 that you should be aware of the behavior of **SELECT *** clauses in regard to attributes with the same name in statements. That is, assuming this **SELECT** clause:

```plaintext
SELECT Amt, *
```

If *** includes an attribute named **Amt**, then the **SELECT** will trigger the EQL error: "Attribute "Amt" is defined more than once."

Likewise in a join:

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

If **a** and **b** both contain an attribute with the same name, then the query triggers the same EQL error as above. It will list one of the attributes that the two sides of the join share. Note that the error message will reference the statement name with the problem.
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:

```eql
define EmployeeTotals AS
  select DimEmployee_FullName AS Name,
         sum(FactSales_SalesAmount) AS Total
  from SaleState
  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:

```eql
return price as select avg(Price) as "Average Price"
```

The space would have to be removed:

```eql
return price as select avg(Price) as AveragePrice
```
FROM clause

You must include a FROM clause in your statement to specify a record source.

A FROM clause is mandatory in a statement and specifies the source of records for an EQL statement, such as from a state name or from a previously-defined statement.

The FROM syntax is:

```
FROM <recSource> [alias]
```

where <recSource> can be:

- The name of previously-defined statement (whether that statement is a DEFINE or a RETURN).
- A state name. Note that FROM does not directly support collection names, but does in essence because the state includes a collection name.
- A JOIN or a CROSS JOIN.

If you omit the FROM clause in your query, the EQL parser returns an error.

Previously-defined statement

You can use the result of a different statement as your record source. In the following example, the first statement (named RepQuarters) 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 (named Quarters) groups those results by quarter:

```
DEFINE RepQuarters AS
SELECT COUNT(TransId) AS NumTrans
FROM SaleState
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 }
...
```

State name

State names can be specified in EQL FROM clauses with this syntax:

```
FROM <statename>
```
This format represents the state with all filters applied (i.e., all the filters that are in the state of the Conversation Service query).

As an example, assume this simple Conversation Service query that uses the EQLQuery type:

```xml
<Request>
  <Language>en</Language>
  <State>
    <Name>WineState</Name>
    <CollectionName>Wines</CollectionName>
    <DataSourceFilter Id="DataFltr">
      <filterString>WineType <> 'Red'</filterString>
    </DataSourceFilter>
    <SelectionFilter Id="SecFltr">
      <filterString>Price > 25</filterString>
    </SelectionFilter>
  </State>
  <EQLConfig Id="WineRecs">
    <EQLQueryString>
      RETURN results AS
      SELECT Price AS prices
      FROM WineState
      GROUP BY prices
    </EQLQueryString>
  </EQLConfig>
</Request>
```

The query works as follows:
1. The DataSourceFilter filter (which is the security filter) first removes any record that has a WineType=Red assignment. In our small data set, only 11 records pass the filter. (Note that WineType must be single-assign or the query will fail.)
2. The SelectionFilter filter then selects any record whose Price assignment is $25 or more. 7 more records are filtered out (from the previous 11 records), leaving 4 records.
3. The FROM clause in the EQL statement references the state named WineState.

Thus, because the FROM clause in the EQL statement references the state named WineState, both filters from the state are applied and the 4 records are returned.

### JOIN clause

JOIN clauses allow records from multiple statements and/or named states 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 (one or both of which can be named states) 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).

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

The syntax of JOIN is as follows:
FROM <statement1> [alias] [INNER,CROSS,LEFT,RIGHT,FULL] JOIN <statement2> [alias] ON (Boolean-expression) [JOIN <statementN> [alias] ON (Boolean-expression)]*

where statement is either a statement or a named state. Note that you can put multiple JOIN clauses under a FROM clause, but there must be exactly one FROM clause in any 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. Note that the INNER keyword can be used only with JOIN, and EQL will throw an error if it is used with the other join types.

• **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.

Keep in mind that if not used correctly, joins can cause the Dgraph to grow beyond available RAM because they can easily create very large results. For example, a CROSS JOIN of a result with 100 records and a result with 200 records would contain 20,000 records. 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.

**INNER JOIN example**

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
  ARB(DimEmployee_FullName) AS Name,
  SUM(FactSales_SalesAmount) AS Total
FROM SaleState
GROUP BY DimEmployee_EmployeeKey, ProductSubcategoryName;

DEFINE SubcategoryTotals AS
SELECT
  SUM(FactSales_SalesAmount) AS Total
FROM SaleState
GROUP BY ProductSubcategoryName;

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

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

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

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

### LEFT JOIN examples

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

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

DEFINE Top5 AS
SELECT
    ARB(Total) AS Total
FROM Totals
GROUP BY ProductSubcategoryName
ORDER BY Total DESC LIMIT 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:

```sql
DEFINE Product AS
SELECT
    ProductAlternateKey AS Key,
    ARB(ProductName) AS Name
FROM SaleState
GROUP BY Key;

DEFINE RegionTrans AS
SELECT
    ProductAlternateKey AS ProductKey,
    FactSales_SalesAmount AS Amount
FROM SaleState
GROUP BY ProductAlternateKey
```
WHERE DimSalesTerritory_SalesTerritoryRegion='United Kingdom';

RETURN Results AS
SELECT 
    Product.Key AS ProductKey,
    ARB(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

FULL JOIN example

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,
    ARB(DimEmployee_FullName) AS Name,
    SUM(FactSales_SalesAmount) AS SalesTotal
FROM SaleState 
GROUP BY Key 
ORDER BY SalesTotal DESC 
PAGE (0,10);

DEFINE TopProducts AS
SELECT 
    ProductAlternateKey AS Key,
    ARB(ProductName) AS Name,
    SUM(FactSales_SalesAmount) AS SalesTotal
FROM SaleState 
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
FROM SaleState 
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)

CROSS JOIN example

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

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: (=, <>, <, <=, >, >=)
- Set operations: such as SUBSET and IS_MEMBER_OF
- Null value evaluation: <attribute> IS {NULL, NOT NULL} (for atomic values) and <attribute> IS {EMPTY, NOT EMPTY} (for sets)
- 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 79.

Aliased attributes (from the SELECT clause) cannot be used in the WHERE clause, because WHERE looks for an attribute in the source. Thus, this example:

```
RETURN results AS
    SELECT
        FactSales_RecordSpec AS id,
        FactSales_ProductKey AS keys
    FROM SaleState
    WHERE id > 5
    ORDER BY keys
```

is invalid and returns the error message:

```
In statement "results": In WHERE clause: The state "Sales" does not have an attribute named "id"
```

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 59.

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
```

```
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) AS JuanTotal
FROM SaleState
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).

The second example also shows how use a per-aggregate **WHERE** clause:

```
SUM(Amount) WHERE (SalesRep = 'Juan Smith') AS JuanTotal
```

For more information on per-aggregate **WHERE** filters, see *Per-aggregation filters on page 46*.

**HAVING clause**

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

The syntax of the **HAVING** clause 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, EMPTY, NOT EMPTY}
- Set operations: such as SUBSET and IS_MEMBER_OF
- 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
FROM SaleState
GROUP BY SalesRep
HAVING SalesTotal > 10000
```

Note that **HAVING** clauses may refer only to attributes defined in the same statement (such as aliased attributes defined by a **SELECT** clause). For example, this is an invalid statement:

```
// Invalid because "Price" is not defined in the statement (i.e., Price is a collection attribute).
Return results AS
SELECT SUM(Price) AS TotalPrices
FROM SaleState
GROUP BY WineType
HAVING Price > 100
```

The invalid statement example would return this error message:
To correct the error, replace the local statement attribute (Price) with an attribute defined in the statement (TotalPrices):

```sql
// Valid because "TotalPrices" is defined in the statement.
RETURN Results AS
SELECT SUM(Price) AS TotalPrices
FROM SaleState
GROUP BY WineType
HAVING TotalPrices > 100
```

**ORDER BY clause**

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:

```sql
ORDER BY <Attr|Exp> [ASC|DESC] [,<Attr|Exp> [ASC|DESC]]*
```

where `Attr|Exp` is either an attribute name or an arbitrary expression. The attribute can be either a single-assign or multi-assign attribute.

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.

An `ORDER BY` clause has the following behavior:

- 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.
- An arbitrary but stable order is used when sorting by sets (multi-assign attributes).
- 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 Price single-assign attribute is totaled and then grouped by the single-assign WineType attribute. The resulting records are sorted by the total amount in descending order:

```sql
RETURN Results AS
SELECT SUM(Price) AS Total
FROM WineState
GROUP BY WineType
ORDER BY Total DESC
```

The result of this statement from a small set of twenty-five records might be:

<table>
<thead>
<tr>
<th>Total</th>
<th>WineType</th>
</tr>
</thead>
<tbody>
<tr>
<td>142.34</td>
<td>Red</td>
</tr>
<tr>
<td>97.97</td>
<td>White</td>
</tr>
<tr>
<td>52.90</td>
<td>Chardonnay</td>
</tr>
<tr>
<td>46.98</td>
<td>Brut</td>
</tr>
<tr>
<td>25.99</td>
<td>Merlot</td>
</tr>
<tr>
<td>21.99</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>16.99</td>
<td>Blanc de Noirs</td>
</tr>
<tr>
<td>14.99</td>
<td>Pinot Noir</td>
</tr>
<tr>
<td></td>
<td>Zinfandel</td>
</tr>
</tbody>
</table>
The Zinfandel bucket is sorted last because it has a NULL value for Price. Note that if the sort order were ASC, Zinfandel would still be last in the result.

**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:

```sql
/* Invalid statement */
DEFINE T1 AS
  SELECT ... AS foo
  FROM SaleState;
RETURN T2 AS
  SELECT ... AS bar
  FROM T1
ORDER BY T1.foo /* not allowed */

/* Valid statement */
DEFINE T1 AS
  SELECT ... AS foo
  FROM SaleState;
RETURN T2 AS
  SELECT ... AS bar
  FROM T1
ORDER BY T1[].foo /* allowed */
```

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

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

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

**Sorting by sets**

As mentioned above, an arbitrary but stable order is used when sorting by sets (multi-assign attributes).

In this example, the Price single-assign attribute is converted to a set and then grouped by the single-assign WineType attribute. The resulting records are sorted by the set in descending order:

```sql
RETURN Results AS
  SELECT SET(Price) AS PriceSet
```
FROM WineState
GROUP BY WineType
ORDER BY PriceSet DESC

The result of this statement from a small set of 25 records might be:

<table>
<thead>
<tr>
<th>PriceSet</th>
<th>WineType</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ 14.99 }</td>
<td>Pinot Noir</td>
</tr>
<tr>
<td>{ 25.99}</td>
<td>Merlot</td>
</tr>
<tr>
<td>{ 22.99, 23.99 }</td>
<td>Brut</td>
</tr>
<tr>
<td>{ 21.99 }</td>
<td>Bordeaux</td>
</tr>
<tr>
<td>{ 25.99}</td>
<td>White</td>
</tr>
<tr>
<td>{ 22.99, 23.99 }</td>
<td>Chardonnay</td>
</tr>
<tr>
<td>{ 16.99 }</td>
<td>Blanc de Noirs</td>
</tr>
<tr>
<td>{ 17.95, 34.95 }</td>
<td>Zinfandel</td>
</tr>
</tbody>
</table>

In this descending order, the Zinfandel bucket is sorted last because it does not have a Price assignment (and thus returns an empty set).

**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 clause**

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.

Note that if `<offset>` is greater than the total number of available records, an empty table is returned. However, if `<offset> + <count>` is greater than the total number of available records, it returns as many records as it can.

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

```
DEFINE Reps AS
FROM ResellerState
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
FROM ResellerState
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
FROM ProductState
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`:

<table>
<thead>
<tr>
<th>PAGE clause</th>
<th>Resulting behavior is the same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAGE (0, 25) PERCENT</td>
<td>PAGE (0, 2)</td>
</tr>
<tr>
<td>PAGE (25, 25) PERCENT</td>
<td>PAGE (2, 1)</td>
</tr>
<tr>
<td>PAGE (50, 25) PERCENT</td>
<td>PAGE (3, 2)</td>
</tr>
<tr>
<td>PAGE (75, 25) PERCENT</td>
<td>PAGE (5, 1)</td>
</tr>
</tbody>
</table>
In EQL, aggregation operations bucket a set of records into a resulting set of aggregated records.

**GROUP/GROUP BY clauses**

- **MEMBERS extension**
- **GROUPING SETS expression**
- **ROLLUP extension**
- **CUBE extension**
- **GROUPING function**
- **COUNT function**
- **COUNT_APPROX**
- **COUNTDISTINCT function**
- **APPROXCOUNTDISTINCT function**
- **Multi-level aggregation**
- **Per-aggregation filters**

**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 `MEMBERS`, `CUBE`, or `GROUPING SETS`) with the `GROUP` and `GROUP BY` clauses. Details on these functions are given later in this section.

**BNF grammar for grouping**

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

```plaintext
GroupClause ::= GROUP | GROUP BY GroupByList | GROUP BY GroupAll
GroupByList ::= GroupByElement | GroupByList , GroupByElement
GroupByElement ::= GroupBySingle | GroupingSets | CubeRollup
```
Aggregation

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 | GroupByMembers
GroupByComposite ::= (GroupByCompositeList)
GroupByCompositeList ::= GroupBySingle | GroupByCompositeList, GroupBySingle
GroupByMembers ::= MEMBERS (Identifier | Identifier.Identifier) AS Identifier

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:

```
SELECT SUM(NumReviews) AS NumberOfReviews
FROM ProductState
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, IN expressions and lookup expressions are valid in this context.

All grouping attributes are part of the result records. In any grouping attribute, NULL values (for single-assign attributes) or empty sets (for multi-assign attributes) are treated like any other value, which means the source record is mapped to result records. For information about user-defined NULL-value handling in EQL, see COALESCE on page 77.

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:
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:

```eql
DEFINE RegionsByYear AS
    GROUP BY Region, Year
```
resulting in the aggregates of the form `{ Region, Year }`, for example:

```json
{ "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:

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

```eql
RETURN Results AS
SELECT SUM(PROMO_COST) AS PR_Cost
FROM SaleState
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:

```eql
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):

```eql
SELECT SUM(Sales) AS Sales2, ...
GROUP BY Sales2
```

### MEMBERS extension

MEMBERS is an extension to GROUP BY that allows grouping by the members of a set.

MEMBERS lets you group by multi-assign attributes. Keep in mind that when grouping by a multi-assign attribute, all rows are preserved (including those with no assignments for the attribute).

### MEMBERS syntax

MEMBERS appears in the GROUP BY clause, using this syntax:

```eql
GROUP BY MEMBERS(<set>) AS <alias> [,MEMBERS(<set2>) AS <alias2>]*
```
Aggregation

where:

- **set** is a set of any set data type (such as `mdex:string-set` or `mdex:long-set`) and must be an attribute reference. For example, *set* can be a multi-assign string attribute from a given collection.

  If **LET** is not used, then **MEMBERS** can only refer to attributes from the source statements or from a collection (i.e., cannot be locally defined). If **LET** is used, then **MEMBERS** can refer to attributes defined in the same statement, as long as those attributes are defined in a **LET** clause, not a **SELECT** clause.

- **alias** is an aliased name, which must be NCName-compliant. In statement results, the aliased name has the same data type as the elements of the set.

As the syntax shows, EQL supports grouping by the members of multiple sets simultaneously. To do this, simply include multiple **MEMBERS** clauses in a **GROUP** list.

The **MEMBERS** form is available in grouping sets, with surface syntax like:

```
GROUP BY ROLLUP(a, b, MEMBERS(c) AS cValue, d)
```

Note that grouping by the members of a set is available in any statement, not just those over a collection (because EQL preserves all values in a set across statement boundaries).

**MEMBERS** data type error message

If an attempt is made to use a single-assign attribute as an argument to **MEMBERS**, an error message is returned similar to this example:

```
Argument to MEMBERS has type mdex:double; only set types are permitted.
```

In this error example, **MEMBERS** was used with a single-assign double attribute (mdex:double), instead of a multi-assign double attribute (mdex:double-set).

**MEMBERS** examples

Assume a small data set of 25 records, with each record having zero, one, or two assignments from the **Body** multi-assign attribute. **WineID** is a single-assign attribute and is the key for the **Wine** collection. This sample query is made:

```
RETURN Results AS
SELECT
  SET(WineID) AS IDs
FROM WineState
GROUP BY MEMBERS(Body) AS bodyType
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>IDs</th>
<th>bodyType</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ 14, 15 }</td>
<td>Supple</td>
</tr>
<tr>
<td>{ 22, 25 }</td>
<td>Firm</td>
</tr>
<tr>
<td>{ 19 }</td>
<td>Fresh</td>
</tr>
<tr>
<td>{ 11, 19, 22, 23, 24, 25, 4, 6, 8 }</td>
<td>Robust</td>
</tr>
<tr>
<td>{ 10, 11, 12, 13, 16, 18, 3, 4, 5, 7, 9 }</td>
<td>Tannins</td>
</tr>
<tr>
<td>{ 10, 12, 13, 16, 18, 3, 5, 7, 9 }</td>
<td>Silky</td>
</tr>
<tr>
<td>{ 1, 17, 2, 20, 21 }</td>
<td></td>
</tr>
</tbody>
</table>

In the results, note that several records contribute to multiple buckets, because they have two **Body** assignments. The last five records in the result have no assignments for the **Body** attribute, but they are not discarded during the grouping and are thus listed with **bodyType** being **NULL**. (Note that using **WineID** allows you to look at the values in the **IDs** sets to determine exactly which input rows contributed to which output...
Aggregation

This second example shows how to group by the members of multiple sets simultaneously. The Body and Score multi-assign attributes are used in the query:

```sql
RETURN Results AS
SELECT
  SET(WineID) as IDs
FROM WineState
WHERE WineType = 'White'
GROUP BY MEMBERS(Body) AS bodyType, MEMBERS(Score) AS scoreValue
```

The result of this query might be:

<table>
<thead>
<tr>
<th>IDs</th>
<th>bodyType</th>
<th>scoreValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ 25 }</td>
<td>Firm</td>
<td>82</td>
</tr>
<tr>
<td>{ 25 }</td>
<td>Firm</td>
<td>84</td>
</tr>
<tr>
<td>{ 19 }</td>
<td>Fresh</td>
<td>88</td>
</tr>
<tr>
<td>{ 25 }</td>
<td>Robust</td>
<td>82</td>
</tr>
<tr>
<td>{ 25 }</td>
<td>Robust</td>
<td>84</td>
</tr>
<tr>
<td>{ 19 }</td>
<td>Robust</td>
<td>88</td>
</tr>
<tr>
<td>{ 20 }</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>{ 20 }</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>{ 21 }</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>{ 21 }</td>
<td></td>
<td>89</td>
</tr>
</tbody>
</table>

Note that the record with WineID=25 contributes to four buckets, corresponding to the cross product of { Firm, Robust } and { 82, 84 }. Records 20 and 21 have assignments for the Score attribute but have no assignments for the Body attribute, and are listed with bodyType being NULL and scoreValue having values.

**Note on MEMBERS interaction with GROUPING SETS**

You should be aware that grouping by set members may interact with GROUPING SETS (including CUBE and ROLLUP) to produce results that at first glance may seem unexpected.

For example, first we make a query that groups only by the ROLLUP extension:

```sql
RETURN Results AS
SELECT
  SUM(Price) AS totalPrice
FROM WineState
GROUP BY ROLLUP(WineType)
```

The result with our data set is:

<table>
<thead>
<tr>
<th>WineType</th>
<th>totalPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanc de Noirs</td>
<td>16.99</td>
</tr>
<tr>
<td>Brut</td>
<td>46.98</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>25.99</td>
</tr>
<tr>
<td>Merlot</td>
<td>21.99</td>
</tr>
<tr>
<td>Bordeaux</td>
<td>52.90</td>
</tr>
<tr>
<td>White</td>
<td>97.97</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>14.99</td>
</tr>
<tr>
<td>Red</td>
<td>142.34</td>
</tr>
<tr>
<td></td>
<td>420.15</td>
</tr>
</tbody>
</table>

We get one row for each WineType, and one summary row at the bottom, which includes records from all of the WineType values. Because SUM is associative, the expected behavior is that the totalPrice summary row
will be equal to the sum of the totalPrice values for all other rows, and in fact the 420.15 result meets that expectation. (Note that the total for White wines is 97.97.)

Then we make a similar query, but selecting only the White wines and grouping with MEMBERS and ROLLUP:

```sql
RETURN Results AS
SELECT
    SUM(Price) AS totalPrice
FROM WineState
WHERE WineType = 'White'
GROUP BY ROLLUP(WineType, MEMBERS(Body) AS bodyType)
```

The result from this second query is:

<table>
<thead>
<tr>
<th>WineType</th>
<th>bodyType</th>
<th>totalPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Firm</td>
<td>43.99</td>
</tr>
<tr>
<td>White</td>
<td>Fresh</td>
<td>20.99</td>
</tr>
<tr>
<td>White</td>
<td>Robust</td>
<td>64.98</td>
</tr>
<tr>
<td>White</td>
<td></td>
<td>32.99</td>
</tr>
<tr>
<td>White</td>
<td></td>
<td>97.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td>97.97</td>
</tr>
</tbody>
</table>

The results show that the correspondence between the summary row and the individual rows is not as expected. One might expect the totalPrice for the 'White' summary row (that is, the row where WineType is White and bodyType is null) to be the sum of the total prices for the (White, Firm), (White, Fresh), and (White, Robust) rows above it.

However, if you add the total prices for the first four rows, you get 162.95, rather than the expected value of 97.97. This discrepancy arises because, when you group by the members of a set, a row can contribute to multiple buckets. In particular, Record 19 has two Body assignments (Fresh and Robust) and therefore contributes to both the (White, Fresh) and (White, Robust) rows, and so its price is in effect double-counted.

EQL effectively computes the 'White' summary row, however, by grouping by WineType (which is a single-assign attribute), so each input row counts exactly once.

**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.

**GROUPING SETS syntax**

The GROUPING SETS syntax is:

```sql
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:

```sql
GROUP BY GROUPING SETS(a, (b), (c, d), ())
```
Multiple grouping sets expressions can exist in the same query.

\[
\text{GROUP BY } a, \text{ GROUPING SETS}(b, c), \text{ GROUPING SETS}((d, e))
\]

is equivalent to:

\[
\text{GROUP BY GROUPING SETS}((a, b, d, e), (a, c, d, e))
\]

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

\[
\text{GROUP} = \text{ GROUP BY}() = \text{ GROUP BY GROUPING SETS}()
\]

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 because duplicate grouping set instances are discarded. For example, these two queries are equivalent:

\[
\text{GROUP BY GROUPING SETS}((x), (x))
\]
\[
\text{GROUP BY GROUPING SETS}((x))
\]

GROUPING SETS example

\[
\text{DEFINE ResellerSales AS}
\]
\[
\text{SELECT SUM(DimReseller_AnnualSales) AS TotalSales,}
\]
\[
\quad \text{ARB(DimReseller_ResellerName) AS RepNames,}
\]
\[
\quad \text{DimReseller_OrderMonth AS OrderMonth}
\]
\[
\text{FROM ResellerState}
\]
\[
\text{GROUP BY OrderMonth;}
\]
\[
\text{RETURN MonthlySales AS}
\]
\[
\text{SELECT AVG(TotalSales) AS AvgSalesPerRep}
\]
\[
\text{FROM ResellerSales}
\]
\[
\text{GROUP BY TotalSales, GROUPING SETS(RepNames), GROUPING SETS(OrderMonth)}
\]

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:

\[
\text{ROLLUP}(a, b, c) = \text{ GROUPING SETS}((a,b,c), (a,b), (a), ())
\]

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. The attributes may be single-assign or multi-assign attributes. ROLLUP can be used on collections.

ROLLUP example

```
DEFINE Resellers AS SELECT
    DimReseller_AnnualSales AS Sales,
    DimGeography_CountryRegionName AS Countries,
    DimGeography_StateProvinceName AS States,
    DimReseller_OrderMonth AS OrderMonth
FROM ResellerState
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 \(2^n\) 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. The attributes may be single-assign or multi-assign attributes. CUBE can be used on collections.

**CUBE example**

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

```sql
DEFINE Resellers AS SELECT
    DimReseller_AnnualSales AS Sales,
    DimGeography_CountryRegionName AS Countries,
    DimGeography_StateProvinceName AS States,
    DimReseller_OrderMonth AS OrderMonth
FROM ResellerState
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:

```sql
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:

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

**GROUPING function**

The GROUPING helper function indicates whether a specified attribute expression in a GROUP BY list is aggregated.

GROUPING is a helping function for GROUPING SETS, CUBE, and ROLLUP. Note that GROUPING cannot be used in a WHERE clause, join condition, inside an aggregate function, or in the definition of a grouping attribute.
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 values.

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:

```sql
GROUPING(attribute)
```

where `attribute` is a single attribute.

**GROUPING example**

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

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)
        }
    }
```

**COUNT function**

The COUNT function returns the number of records that have a value for an attribute.

The COUNT function counts the number of records that have non-NULL values in a field for each GROUP BY result. COUNT can be used with both multi-assign attributes (sets) and single-assign attributes.
For multi-assign attributes, the COUNT function counts all non-NULL sets in the group. Note that because sets are never NULL but can be empty, COUNT will also count a record with an empty set (that is, an empty set is returned for any record that does not have an assignment for the specified multi-assign attribute). See the second example below for how to ignore empty sets from the results.

The syntax of the COUNT function is:

```
COUNT(<attribute>)
```

where attribute is either a multi-assign or single-assign attribute.

### COUNT examples

The following records include the single-assign Size attribute and the multi-assign Color attribute:

- 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
FROM ProductState
GROUP BY Size
```

The statement result is:

<table>
<thead>
<tr>
<th>Record</th>
<th>Size</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record 1: Size=small</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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 4, because Records 1-3 have Color assignments (and thus return non-empty sets) and Record 4 does not have a Color assignment (and an empty set is returned).

If you are using COUNT with a multi-assign attribute and want to exclude empty sets, use a per-aggregate WHERE clause with the IS NOT EMPTY function, as in this example:

```
RETURN result AS
SELECT COUNT(Color) WHERE (Color IS NOT EMPTY) AS Total
FROM ProductState
GROUP BY Size
```

This statement result is:

<table>
<thead>
<tr>
<th>Record</th>
<th>Size</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record 1: Size=small</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

because the empty set for Record 4 is not counted.

### COUNT(1) format

The COUNT(1) syntax returns a count of all records (including those with NULL values) in a specific collection. For example, you can get the number of data records in your Sales collection as follows:

```
RETURN Results AS
SELECT COUNT(1) AS recordCount
FROM SalesState
GROUP
```

The statement result should be an integer that represents the total number of data records.
COUNT_APPROX returns the most frequent refinements.

COUNT_APPROX is similar to the COUNT function except that it is allowed to produce imprecise results in certain circumstances. Like COUNT, the COUNT_APPROX function counts the number of records that have non-NULL values in a field for each GROUP BY result.

The syntax of the COUNT_APPROX function is:

```
COUNT_APPROX(<attribute>)
```

where attribute is either a multi-assign or single-assign attribute. You can also use the COUNT_APPROX(1) format.

The COUNT_APPROX function uses a FrequentK pattern-matching algorithm that calculates a set of refinements. Specifically, it reports the most frequent values. By using the PAGE function, you can indicate the frequency range (as illustrated in the example below).

COUNT_APPROX works best when the distribution is skewed so that a small set of values appear very frequently. However, if the FrequentK pattern matching fails to produce any results, then COUNT_APPROX falls back to using the same implementation as the COUNT function (which does not use the FrequentK algorithm). When running in the FrequentK pattern-matching mode, COUNT_APPROX may return imprecise results; however, its accuracy is precise if it falls back to COUNT mode.

COUNT_APPROX example

In this example, COUNTRY is a single-assign attribute containing country names:

```
RETURN Results AS
    SELECT
        COUNT_APPROX(COUNTRY) AS Approx
    FROM SalesData
    WHERE COUNTRY IS NOT NULL
    GROUP BY COUNTRY
    ORDER BY Approx DESC
    PAGE(0, 10)
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>Approx</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>81970</td>
<td>USA</td>
</tr>
<tr>
<td>1590</td>
<td>GERMANY</td>
</tr>
<tr>
<td>1353</td>
<td>JAPAN</td>
</tr>
<tr>
<td>667</td>
<td>KOREA</td>
</tr>
<tr>
<td>598</td>
<td>ENGLAND</td>
</tr>
<tr>
<td>585</td>
<td>ITALY</td>
</tr>
<tr>
<td>546</td>
<td>CANADA</td>
</tr>
<tr>
<td>242</td>
<td>GUAM</td>
</tr>
<tr>
<td>203</td>
<td>COLOMBIA</td>
</tr>
<tr>
<td>176</td>
<td>SPAIN</td>
</tr>
</tbody>
</table>

COUNTDISTINCT function

The COUNTDISTINCT function counts 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 be used for both single-assign and multi-assigned attributes.
Note that because sets are never NULL but can be empty, \textit{COUNTDISTINCT} will also evaluate a record with an empty set (that is, an empty set is returned for any record that does not have an assignment for the specified multi-assign attribute). See the second example below for how to ignore empty sets from the results.

The syntax of the \textit{COUNTDISTINCT} function is:

\begin{verbatim}
COUNTDISTINCT(<attribute>)
\end{verbatim}

where \textit{attribute} is either a multi-assign or single-assign attribute.

\section*{COUNTDISTINCT example}

The following records include the single-assign Size attribute and the multi-assign Color attribute:

\begin{verbatim}
Record 1: Size=small, Color=red
Record 2: Size=small, Color=blue
Record 3: Size=small, Color=red
Record 4: Size=small
\end{verbatim}

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

\begin{verbatim}
RETURN Result AS
SELECT COUNTDISTINCT (Color) as Total
FROM ProductState
GROUP BY Size
\end{verbatim}

The statement result is:

\begin{verbatim}
Record 1: Size=small, Total=3
\end{verbatim}

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 there are two non-empty sets with unique values for the Color attribute (red and blue), and an empty set is returned for Record 4.

If you are using \textit{COUNTDISTINCT} with a multi-assign attribute and want to exclude empty sets, use a \textit{WHERE} clause with the \textit{IS NOT EMPTY} function, as in this example:

\begin{verbatim}
RETURN Result AS
SELECT COUNTDISTINCT(Color) WHERE (Color IS NOT EMPTY) AS Total
FROM ProductState
GROUP BY Size
\end{verbatim}

This statement result is:

\begin{verbatim}
Record 1: Size=small, Total=2
\end{verbatim}

because the empty set for Record 4 is not counted.

\section*{APPROXCOUNTDISTINCT function}

The \textit{APPROXCOUNTDISTINCT} function counts the number of distinct values for an attribute.

\textit{APPROXCOUNTDISTINCT} is similar to the \textit{COUNTDISTINCT} function except that it is allowed to produce an approximation for the number of distinct values in certain circumstances. The \textit{APPROXCOUNTDISTINCT} function returns the number of unique values in a field for each \textit{GROUP BY} result. \textit{APPROXCOUNTDISTINCT} can be used for both single-assign and multi-assigned attributes.

The \textit{APPROXCOUNTDISTINCT} function uses the HyperLogLog algorithm to calculate the a set of refinements. If the number of distinct values is low, then the results will be accurate; if the number of distinct values is high, the results will be an approximation.
APPROXCOUNTDISTINCT will also evaluate a record with an empty set (that is, an empty set is returned for any record that does not have an assignment for the specified multi-assign attribute).

**APPROXCOUNTDISTINCT syntax**

The syntax of the APPROXCOUNTDISTINCT function is:

```
APPROXCOUNTDISTINCT(<attribute>)
```

where `attribute` is either a multi-assign or single-assign attribute.

**APPROXCOUNTDISTINCT example**

Assume the following nine records that are of WineType=Red (where WineType is a single-assign attribute). Each record includes one or two assignments for the multi-assign Body attribute:

<table>
<thead>
<tr>
<th>Body</th>
<th>WineID</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Silky, Tannins } 3</td>
<td></td>
</tr>
<tr>
<td>{ Robust, Tannins } 4</td>
<td></td>
</tr>
<tr>
<td>{ Silky, Tannins } 5</td>
<td></td>
</tr>
<tr>
<td>{ Robust }          6</td>
<td></td>
</tr>
<tr>
<td>{ Robust }          8</td>
<td></td>
</tr>
<tr>
<td>{ Silky, Tannins } 9</td>
<td></td>
</tr>
<tr>
<td>{ Silky, Tannins } 12</td>
<td></td>
</tr>
<tr>
<td>{ Silky, Tannins } 16</td>
<td></td>
</tr>
<tr>
<td>{ Silky, Tannins } 18</td>
<td></td>
</tr>
</tbody>
</table>

The following statement returns the number of different values for the Body attribute in the WineType=Red records:

```sql
RETURN Result AS
SELECT APPROXCOUNTDISTINCT (Body) AS Total
FROM WineState
WHERE WineType = 'Red'
GROUP BY WineType
```

The statement result is:

```
Total=3, WineType=Red
```

For this group, the value of Total is 3 because there are three non-empty sets with unique values for the Body attribute:

- One set for Records 3, 5, 9, 12, 16, and 18, each of which has the "Silky" and "Tannins" assignments for Body.
- One set for Records 6 and 8, each of which has the "Robust" assignment for Body.
- One set for Record 4, which has the "Robust" and "Tannins" assignments for Body.

Thus, there are three sets of distinct values for the Body attribute, when grouped by the WineType attribute.

**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.

```sql
DEFINE DealCount AS
SELECT COUNT(TransId) AS NumDeals
FROM SaleState
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.

Per-aggregate `WHERE` filters are indeed applied pre-aggregation. The reason is that if it is delayed until post-aggregation, the implementation may not necessarily have access to all of the columns that it needs.

The per-aggregation syntax is:

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

For example:

```sql
RETURN NetSales AS
SELECT
  SUM(Amount) WHERE (Type='Sale') AS SalesTotal,
  SUM(Amount) WHERE (Type='Return') AS ReturnTotal,
  ARB(SalesTotal - ReturnTotal) AS Total
FROM SaleState
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.
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
Operator precedence rules
Literals
Functions and operators
Using EQL results to compose follow-on queries
Using LOOKUP expressions for inter-statement references
ARB
BETWEEN
CASE
COALESCE
CORRELATION
HAS_REFINEMENTS
IN
PERCENTILE
RECORD_IN_FAST_SAMPLE

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

<table>
<thead>
<tr>
<th>EQL data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdex:boolean</td>
<td>Represents a Boolean value (TRUE or FALSE). Used for atomic values (from single-assign Boolean attributes).</td>
</tr>
<tr>
<td>mdex:boolean-set</td>
<td>Represents a Boolean value (TRUE or FALSE). Used for sets (from multi-assign Boolean attributes).</td>
</tr>
<tr>
<td>mdex:dateTime</td>
<td>Represents a date and time to a resolution of milliseconds. Used for atomic values (from single-assign dateTime attributes).</td>
</tr>
<tr>
<td>EQL data type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mdex:dateTime-set</td>
<td>Represents a date and time to a resolution of milliseconds. Used for sets (from multi-assign dateTime attributes).</td>
</tr>
<tr>
<td>mdex:double</td>
<td>Represents a floating point number. Used for atomic values (from single-assign double attributes).</td>
</tr>
<tr>
<td>mdex:double-set</td>
<td>Represents a floating point number. Used for sets (from multi-assign double attributes).</td>
</tr>
<tr>
<td>mdex:duration</td>
<td>Represents a length of time with a resolution of milliseconds. Used for atomic values (from single-assign duration attributes).</td>
</tr>
<tr>
<td>mdex:duration-set</td>
<td>Represents a length of time with a resolution of milliseconds. Used for sets (from multi-assign duration attributes).</td>
</tr>
<tr>
<td>mdex:geocode</td>
<td>Represents a latitude and longitude pair. Used for atomic values (from single-assign geocode attributes).</td>
</tr>
<tr>
<td>mdex:geocode-set</td>
<td>Represents a latitude and longitude pair. Used for sets (from multi-assign geocode attributes).</td>
</tr>
<tr>
<td>mdex:long</td>
<td>Represents a 64-bit integer. Used for atomic values (from single-assign 32-bit integer attributes and single-assign 64-bit long attributes).</td>
</tr>
<tr>
<td></td>
<td>Note that 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 (i.e., mdex:long data type). This means that if you query an attribute that has a 32-bit integer value, it will appear as a long (64-bit value) in EQL results.</td>
</tr>
<tr>
<td>mdex:string</td>
<td>Represents character strings. Used for atomic values (from single-assign string attributes).</td>
</tr>
<tr>
<td>mdex:string-set</td>
<td>Represents character strings. Used for sets (from multi-assign string attributes).</td>
</tr>
<tr>
<td>mdex:time</td>
<td>Represents the time of day to a resolution of milliseconds. Used for atomic values (from single-assign time attributes).</td>
</tr>
<tr>
<td>mdex:time-set</td>
<td>Represents the time of day to a resolution of milliseconds. Used for sets (from multi-assign time attributes).</td>
</tr>
</tbody>
</table>
Operator precedence rules

EQL enforces the following precedence rules for operators.

The rules are listed in descending order:

1. Parentheses (as well as brackets in lookup expressions and IN expressions). Note that you can freely add parentheses any time you want to impose an alternative precedence or to make precedence clearer.

2. */

3. + -

4. = <> < > <= >=

5. IS (IS NULL, IS NOT NULL, IS EMPTY, IS NOT EMPTY)

6. IN

7. BETWEEN

8. NOT

9. AND

10. OR

Except for IN, the binary operators are left-associative, as are all of the JOIN operators. IN (for set membership) is not associative (for example, writing x IN y IN z results in a syntax error.)

Comparisons with sets

When comparing values against sets (multi-assign data), you must use the appropriate set functions and expressions.

For example, if Price is a single-assign double attribute, then this syntax is correct:

```
RETURN Results AS
SELECT Price AS prices
FROM ProductsState
WHERE Price > 20
```

However, if Score is a multi-assign integer attribute, then this syntax will fail:

```
RETURN Results AS
SELECT Score AS ratings
FROM ProductsState
WHERE Score > 80
```

The error message will be:

```
In statement "Results": in WHERE clause: The comparison operators are not defined on arguments of types mdex:long-set and mdex:long
```

The error message means that Score is a set (an mdex:long-set data type) and therefore cannot be compared to an integer (80, which is an mdex:long data type).

You therefore must re-write the query, as in this example:

```
RETURN Results AS
SELECT Score AS Ratings
FROM ProductsState
WHERE SOME x IN Score SATISFIES (x > 80)
```
This example uses an existential quantifier expression.

**Literals**

This section discusses how literals are used in EQL.

*Character handling*

*Handling of upper- and lower-case*

*Handling NULL attribute values*

*Integer type promotion*

*Handling of NaN, inf, and -inf results*

**Character handling**

EQL accepts all Unicode characters.

\[
\text{Literal} ::= \text{StringLiteral} | \text{NumericLiteral}
\]

<table>
<thead>
<tr>
<th>Literal type</th>
<th>Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String literals</strong></td>
<td>String literals must be surrounded by single quotation marks.</td>
</tr>
<tr>
<td></td>
<td>Embedded single quotes and backslashes must be escaped by backslashes.</td>
</tr>
<tr>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td>'jim'</td>
</tr>
<tr>
<td></td>
<td>'âlèx's house'</td>
</tr>
<tr>
<td><strong>Numeric literals</strong></td>
<td>Numeric literals can be integers or floating point numbers.</td>
</tr>
<tr>
<td></td>
<td>Numeric literals cannot be surrounded by single quotation marks.</td>
</tr>
<tr>
<td></td>
<td>Numeric literals do not support exponential notation.</td>
</tr>
<tr>
<td></td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>.34</td>
</tr>
<tr>
<td><strong>Boolean literal</strong></td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td></td>
<td>Boolean literals cannot be surrounded by single quotation marks.</td>
</tr>
<tr>
<td><strong>Literals of structured types</strong></td>
<td>Literals of structured types must use appropriate conversions, as shown</td>
</tr>
<tr>
<td>(such as Date, Time, or Geocode)</td>
<td>in the following example:</td>
</tr>
<tr>
<td></td>
<td>RETURN Result AS</td>
</tr>
<tr>
<td></td>
<td>SELECT TO_GEOCODE(45.0, 37.0) AS Geocode,</td>
</tr>
<tr>
<td></td>
<td>TO_DATETIME(’2012-11-21T08:22:00Z’) AS Timestamp</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
### Literal type

<table>
<thead>
<tr>
<th>Handling</th>
</tr>
</thead>
</table>

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/](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. For example, if an attribute name contains a hyphen (which is a valid NCName), then the attribute name must be enclosed in double quotation marks in statements. Otherwise, the hyphen will be treated as the subtraction operator by the EQL parser.

- 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
- Attribute references

Reserved words are case insensitive.
**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:

<table>
<thead>
<tr>
<th>Type of operation</th>
<th>How EQL handles NULL values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operations and non-aggregating functions</td>
<td>The value of any operation on a NULL value is also defined as NULL.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Aggregating functions</td>
<td>EQL ignores records with NULL values.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>If all 10 records have a NULL Price, then most aggregations, such as SUM(Price), also result in NULL values.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Boolean operators</td>
<td>See <em>Boolean operators on page 70.</em></td>
</tr>
<tr>
<td>Grouping expressions</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Expressions

<table>
<thead>
<tr>
<th>Type of operation</th>
<th>How EQL handles NULL values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filters</td>
<td>When doing a comparison against a specific value, the NULL value will not match the specified filter, except for the IS NULL filter. Note that:</td>
</tr>
<tr>
<td></td>
<td>• Filters used directly on collections have the same semantics as filters on intermediate results.</td>
</tr>
<tr>
<td></td>
<td>• NOT(x=y) is always equivalent to x&lt;&gt;y for all filters. For example, if record A has price 5, and record B has no price value, then:</td>
</tr>
<tr>
<td></td>
<td>• WHERE price = 5 matches A</td>
</tr>
<tr>
<td></td>
<td>• WHERE NOT(price &lt;&gt; 5) matches A</td>
</tr>
<tr>
<td></td>
<td>• WHERE price &lt;&gt; 5 matches neither A nor B</td>
</tr>
<tr>
<td></td>
<td>• WHERE NOT(price = 5) matches neither A nor B</td>
</tr>
<tr>
<td></td>
<td>• WHERE price = 99 matches neither A nor B</td>
</tr>
<tr>
<td></td>
<td>• WHERE NOT(price &lt;&gt; 99) matches neither A nor B</td>
</tr>
<tr>
<td></td>
<td>• WHERE price &lt;&gt; 99 matches A</td>
</tr>
<tr>
<td></td>
<td>• WHERE NOT(price = 99) matches A</td>
</tr>
<tr>
<td>Sorting</td>
<td>For any sort order specified, EQL returns:</td>
</tr>
<tr>
<td></td>
<td>1. Normal results</td>
</tr>
<tr>
<td></td>
<td>2. Records for a NaN value</td>
</tr>
<tr>
<td></td>
<td>3. Records with a NULL value</td>
</tr>
</tbody>
</table>

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

### Integer type promotion

In some cases, EQL supports automatic value promotion of integers to doubles when there is no risk of loss of information.

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

\[
\text{COALESCE}(\text{Quantity}, \text{SingleOrder})
\]

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

\[
\text{COALESCE}(\text{Quantity}, \text{TO_INTEGER}(\text{SingleOrder}))
\]

**Handling of NaN, inf, and -inf results**

Operations in EQL adhere to the conventions for Not a Number (NaN), inf, and -inf defined by the IEEE 754 2008 standard for handling floating point numbers.

In cases where 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:

<table>
<thead>
<tr>
<th>Type of operation</th>
<th>How EQL handles NaN, inf, and -inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic operations</td>
<td>Arithmetic operations with NaN values result in NaN values.</td>
</tr>
<tr>
<td>Filters</td>
<td>NaN values do not pass filters (except for &lt;&gt;). Any other comparison involving a NaN value is false.</td>
</tr>
<tr>
<td>Sorting</td>
<td>For any sort order specified, EQL returns: 1. Normal records 2. Records with a NaN value 3. Records with a NULL value</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ASC</th>
<th>DESC</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inf</td>
<td>+inf</td>
</tr>
<tr>
<td>-4</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>-4</td>
</tr>
<tr>
<td>+inf</td>
<td>-inf</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Functions and operators**

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

**Important:** With some exceptions, all the functions and operators mentioned in this chapter work only on atomic data types. That is, they are not supported with sets. The exceptions are:

- ARB
- COUNT
- COUNT_APPROX
- COUNTDISTINCT
- APPROXCOUNTDISTINCT
- HAS_REFINEMENTS

For information on the set functions, see *Sets and Multi-assign Data on page 82.*

**Numeric functions**

**Aggregation functions**

**Geocode functions**
### Numeric functions

EQL supports the following numeric functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description and Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>addition</strong></td>
<td>The addition operator (+).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT NortheastSales + SoutheastSales AS EastTotalSales</code></td>
</tr>
<tr>
<td><strong>subtraction</strong></td>
<td>The subtraction operator (-).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT SalesRevenue - TotalCosts AS Profit</code></td>
</tr>
<tr>
<td><strong>multiplication</strong></td>
<td>The multiplication operator (*).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT Price * 0.7 AS SalePrice</code></td>
</tr>
<tr>
<td><strong>division</strong></td>
<td>The division operator (/).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT YearTotal / 4 AS QuarterAvg</code></td>
</tr>
<tr>
<td><strong>ABS</strong></td>
<td>Returns the absolute value of ( n ).</td>
</tr>
<tr>
<td></td>
<td>If ( n ) is 0 or a positive integer, returns ( n ).</td>
</tr>
<tr>
<td></td>
<td>Otherwise, ( n ) is multiplied by -1.</td>
</tr>
<tr>
<td></td>
<td><code>SELECT ABS(-1) AS one</code></td>
</tr>
<tr>
<td><strong>RESULT:</strong></td>
<td>( one = 1 )</td>
</tr>
<tr>
<td><strong>CEIL</strong></td>
<td>Returns the smallest integer value not less than ( n ).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT CEIL(123.45) AS x, CEIL(32) AS y, CEIL(-123.45) AS z</code></td>
</tr>
<tr>
<td><strong>RESULT:</strong></td>
<td>( x = 124, y = 32, z = 123 )</td>
</tr>
<tr>
<td><strong>EXP</strong></td>
<td>Exponentiation, where the base is ( e ).</td>
</tr>
<tr>
<td></td>
<td>Returns the value of ( e ) (the base of natural logarithms) raised to the power ( n ).</td>
</tr>
<tr>
<td></td>
<td><code>SELECT EXP(1.0) AS baseE</code></td>
</tr>
<tr>
<td><strong>RESULT:</strong></td>
<td>( baseE = e^{1.0} = 2.71828182845905 )</td>
</tr>
<tr>
<td>Function</td>
<td>Description and Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><strong>FLOOR</strong></td>
<td>Returns the largest integer value not greater than ( n ).</td>
</tr>
<tr>
<td></td>
<td>( \text{SELECT FLOOR}(123.45) \text{ AS } x, \text{ FLOOR}(32) \text{ AS } y, \text{ FLOOR}(-123.45) \text{ AS } z )</td>
</tr>
<tr>
<td></td>
<td><strong>RESULT:</strong> ( x = 123 ), ( y = 32 ), ( z = 124 )</td>
</tr>
<tr>
<td><strong>LN</strong></td>
<td>Natural logarithm. Computes the logarithm of its single argument, the base of which is ( e ).</td>
</tr>
<tr>
<td></td>
<td>( \text{SELECT LN}(1.0) \text{ AS } \text{baseE} )</td>
</tr>
<tr>
<td></td>
<td><strong>RESULT:</strong> ( \text{baseE} = e^{1.0} = 0 )</td>
</tr>
<tr>
<td><strong>LOG</strong></td>
<td>Logarithm. ( \log(n, m) ) takes two arguments, where ( n ) is the base, and ( m ) is the value you are taking the logarithm of.</td>
</tr>
<tr>
<td></td>
<td>( \log(10,1000) = 3 )</td>
</tr>
<tr>
<td><strong>MOD</strong></td>
<td>Modulo. Returns the remainder of ( n ) divided by ( m ).</td>
</tr>
<tr>
<td></td>
<td>( \text{Mod}(10,3) = 1 )</td>
</tr>
<tr>
<td></td>
<td>EQL uses the ( fmod ) floating point remainder, as defined in the C/POSIX standard.</td>
</tr>
<tr>
<td><strong>ROUND</strong></td>
<td>Returns a number rounded to the specified decimal place.</td>
</tr>
<tr>
<td></td>
<td>The unary (one argument) version takes only one argument (the number to be rounded) and drops the decimal (non-integral) portion of the input. For example:</td>
</tr>
<tr>
<td></td>
<td>( \text{ROUND}(8.2) \text{ returns } 8 ) ( \text{ROUND}(8.7) \text{ returns } 9 )</td>
</tr>
<tr>
<td></td>
<td>The binary (two argument) 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). The binary version always returns a double:</td>
</tr>
<tr>
<td></td>
<td>• Positive second arguments correspond to the number of places that must be returned after the decimal point. For example:</td>
</tr>
<tr>
<td></td>
<td>( \text{ROUND}(123.4567, 3) \text{ returns } 123.457 )</td>
</tr>
<tr>
<td></td>
<td>• Negative second arguments correspond to the number of places that must be returned before the decimal point. For example:</td>
</tr>
<tr>
<td></td>
<td>( \text{ROUND}(123.4, -3) \text{ returns } 0 ) ( \text{ROUND}(1234.56, -3) \text{ returns } 1000 )</td>
</tr>
<tr>
<td><strong>SIGN</strong></td>
<td>Returns the sign of the argument as -1, 0, or 1, depending on whether ( n ) is negative, zero, or positive. The result is always a double.</td>
</tr>
<tr>
<td></td>
<td>( \text{SELECT SIGN}(-12) \text{ AS } x, \text{ SIGN}(0) \text{ AS } y, \text{ SIGN}(12) \text{ AS } z )</td>
</tr>
<tr>
<td></td>
<td><strong>RESULT:</strong> ( x = -1 ), ( y = 0 ), ( z = 1 )</td>
</tr>
<tr>
<td>Function</td>
<td>Description and Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
</tbody>
</table>
| SQRT | Returns the nonnegative square root of \( n \) as a \texttt{mdex:double} type.  
\texttt{SELECT SQRT(9) AS x}  
\textbf{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 (one argument) version drops the decimal (non-integral) portion of the input. For example:  
\texttt{SELECT TRUNC(3.14159265) AS x}  
\textbf{RESULT:} \( x = 3 \)  
The binary (two argument) version allows you to set the number of spaces at which the number is truncated. The binary version always returns a double. For example:  
\texttt{SELECT TRUNC(3.14159265, 3) AS y}  
\textbf{RESULT:} \( y = 3.141 \) |
| SIN | The sine of \( n \), where the angle of \( n \) is in radians.  
\texttt{SIN(3.14159/6) = 0.499999616987256} |
| COS | The cosine of \( n \), where the angle of \( n \) is in radians.  
\texttt{COS(3.14159/3) = 0.500000766025195} |
| TAN | The tangent of \( n \), where the angle of \( n \) is in radians.  
\texttt{TAN(3.14159/4) = 0.999998673205984} |
| POWER | Returns the value (as a double) of \( n \) raised to the power of \( m \).  
\texttt{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 \texttt{TRUE}/\texttt{FALSE} to 1/0. |
## Aggregation functions

EQL supports the following aggregation functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARB</td>
<td>Selects an arbitrary but consistent value from the set of values in a field. Works on both multi-assign attributes (sets) and single-assign attributes.</td>
</tr>
<tr>
<td>AVG</td>
<td>Computes the arithmetic mean value for a field.</td>
</tr>
<tr>
<td>CORRELATION</td>
<td>Computes the correlation coefficient between two numeric fields.</td>
</tr>
<tr>
<td>COUNT</td>
<td>Counts the number of records with valid non-NULL values in a field for each <code>GROUP BY</code> result. Works on both multi-assign attributes (sets) and single-assign attributes.</td>
</tr>
<tr>
<td>COUNT_APPROX</td>
<td>Counts the most frequent refinements. Works on both multi-assign attributes (sets) and single-assign attributes.</td>
</tr>
<tr>
<td>COUNTDISTINCT</td>
<td>Counts the number of unique, valid non-NULL values in a field for each <code>GROUP BY</code> result. Works on both multi-assign attributes (sets) and single-assign attributes.</td>
</tr>
<tr>
<td>APPROXCOUNTDISTINCT</td>
<td>Counts the number of unique, valid non-NULL values in a field for each <code>GROUP BY</code> result. Works on both multi-assign attributes (sets) and single-assign attributes.</td>
</tr>
<tr>
<td>HAS_REFINEMENTS</td>
<td>Determines whether a specific attribute has non-implicit refinements.</td>
</tr>
<tr>
<td>MAX</td>
<td>Finds the maximum value for a field.</td>
</tr>
<tr>
<td>MIN</td>
<td>Finds the minimum value for a field.</td>
</tr>
</tbody>
</table>
| MEDIAN            | Finds the median value for a field. (Note that `PAGE PERCENT` provides overlapping functionality). If the argument is an integer, a double is always returned.  
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:

```sql
RETURN results AS SELECT
    MEDIAN(a) AS med
FROM SaleState
GROUP
```

produces the mean of the two elements in the middle of the sorted set, or 2.5. |
<p>| PERCENTILE        | Computes the percentile for a field.                                                                                                                                                                         |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RECORD_IN_FAST_SAMPLE</td>
<td>Returns a sample of the records in the named state.</td>
</tr>
<tr>
<td>STDDEV</td>
<td>Computes the standard deviation for a field.</td>
</tr>
<tr>
<td>STRING_JOIN</td>
<td>Creates a single string containing all the values of a string attribute.</td>
</tr>
<tr>
<td>SUM</td>
<td>Computes the sum of field values.</td>
</tr>
<tr>
<td>VARIANCE</td>
<td>Computes the variance (that is, the square of the standard deviation) for a field.</td>
</tr>
</tbody>
</table>

**MIN and MAX results ordering**

The **MIN** and **MAX** functions work with int, double, dateTime, duration, Boolean, and string fields, as follows:

- For int and double values, **MIN** finds the numerically smallest integer or double, while **MAX** finds the largest integer or double.
- For dateTime values, **MIN** finds the earliest date while **MAX** finds the latest date.
- For duration values, **MIN** finds the shortest time duration date while **MAX** finds the longest time duration. Note that a negative duration is considered to be less than a positive duration.
- For Boolean values, both **MIN** and **MAX** consider FALSE to be less than TRUE (if the data set has both values assigned). If the data set has only Boolean type assigned, then that value is returned by both functions.
- For string values, both functions use the lexicographical ordering (for example, "89" < "9" < "90" < "ab" < "xy"). In this example, **MIN** would return "89" while **MAX** would return "xy".

**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 a lexicographical 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** attribute is of type string and contains names of regions, while the **N_NAME** 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
FROM ProductState
GROUP
```

The query returns the region and country names delimited by commas:
Note: The Regions delimiter includes a space while the Nations delimiter does not. That is, if you want a space between the output terms, you must specify it in the delimiter.

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATITUDE(mdex:geocode)</td>
<td>Returns the latitude of a geocode as a floating-point number.</td>
</tr>
<tr>
<td>LONGITUDE(mdex:geocode)</td>
<td>Returns the longitude of a geocode as a floating-point number.</td>
</tr>
<tr>
<td>DISTANCE(mdex:geocode,</td>
<td>Returns the distance (in kilometers) between the two geocodes, using the</td>
</tr>
<tr>
<td>mdex:geocode)</td>
<td>haversine formula.</td>
</tr>
<tr>
<td>TO_GEOCODE(mdex:double,</td>
<td>Creates a geocode from the given latitude and longitude.</td>
</tr>
<tr>
<td>mdex:double)</td>
<td></td>
</tr>
</tbody>
</table>

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
FROM ProductState
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. The Dgraph conforms to this recommendation.
The following table summarizes the supported date and time functions:

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

Note that using CURRENT_DATE, CURRENT_TIMESTAMP, SYSDATE, or SYSTIMESTAMP affects performance because those functions are not cached. The other functions in the table are cached.

The following table summarizes supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Return Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (+</td>
<td>-) duration</td>
</tr>
<tr>
<td>dateTime (+</td>
<td>-) duration</td>
</tr>
<tr>
<td>time - time</td>
<td>duration</td>
</tr>
<tr>
<td>dateTime - dateTime</td>
<td>duration</td>
</tr>
<tr>
<td>duration (+</td>
<td>-) duration</td>
</tr>
<tr>
<td>duration &quot;/&quot; duration</td>
<td>duration</td>
</tr>
<tr>
<td>duration /duration</td>
<td>double</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>Obtains current date and time in GMT.</td>
</tr>
<tr>
<td>SYSTIMESTAMP</td>
<td>Obtains current date and time in server time zone.</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>Obtains current date in GMT.</td>
</tr>
<tr>
<td>SYSDATE</td>
<td>Obtains system date in server time zone.</td>
</tr>
</tbody>
</table>

**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
FROM EmployeeState
GROUP
```

Constructing date and time values

EQL provides functions to construct a timestamp representing time, date, or duration using an expression. If the expression is a string, it must be in a certain format. If the format is invalid or the value is out of range, it results in NULL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_TIME</td>
<td>Constructs a timestamp representing time.</td>
<td>&lt;TimeStringFormat&gt; ::= hh:mm:ss[.sss]((+-) hh:mm</td>
</tr>
<tr>
<td>TO_DATETIME</td>
<td>Constructs a timestamp representing date and time.</td>
<td>See the section below for the syntax of this function's string interface, date-only numeric interface, and date-time numeric interface.</td>
</tr>
</tbody>
</table>
### Function: TO_DURATION

Constructs a timestamp representing duration.

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{DurationStringFormat}&gt;::=)</td>
</tr>
<tr>
<td>([\text{timestamp}]-\text{P}{\text{&lt;Days&gt;}}}{\text{&lt;Hours&gt;}}}{\text{&lt;Minutes&gt;}}}{\text{&lt;Seconds&gt;}})</td>
</tr>
<tr>
<td>(&lt;\text{Days}&gt;::=\text{&lt;Integer&gt;}\ D)</td>
</tr>
<tr>
<td>(&lt;\text{Hours}&gt;::=\text{&lt;Integer&gt;}\ H)</td>
</tr>
<tr>
<td>(&lt;\text{Minutes}&gt;::=\text{&lt;Integer&gt;}\ M)</td>
</tr>
<tr>
<td>(&lt;\text{Seconds}&gt;::=\text{&lt;Integer&gt;}\ .\text{&lt;Integer&gt;}\ S)</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Normalized value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_DATETIME('2012-03-21T16:00:00.000+02:00')</td>
<td>2012-03-21T14:00:00.000Z</td>
</tr>
<tr>
<td>TO_DATETIME('2012-12-31T20:00:00.000-06:00')</td>
<td>2013-01-01T02:00:00.000Z</td>
</tr>
<tr>
<td>TO_DATETIME('2012-06-15T20:00:00.000Z')</td>
<td>2012-06-15T20:00:00.000Z</td>
</tr>
<tr>
<td>TO_TIME('23:00:00.000+03:00')</td>
<td>20:00:00.000Z</td>
</tr>
<tr>
<td>TO_TIME('15:00:00.000-10:00')</td>
<td>01:00:00.000Z</td>
</tr>
</tbody>
</table>

### 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.
Expressions

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_DURATI0N) and time zone (TO_TZ) constructs can be used, as shown below in the examples.

Examples of the numeric interface signatures are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_DATETIME(2012, 9, 22)</td>
<td>2012-09-22T00:00:00.000Z</td>
</tr>
<tr>
<td>TO_DATETIME(2012, 9, 22, 23, 15, 50, 500)</td>
<td>2012-09-22T23:15:50.500Z</td>
</tr>
<tr>
<td>TO_DATETIME(2012, 9, 22, 23, 15, 50, 500) + TO_DURATI0N(1000)</td>
<td>2012-09-22T24:15:50.500Z</td>
</tr>
<tr>
<td>TO_TZ(TO_DATETIME(2012, 9, 22, 23, 15, 50, 500), 'America/New_York')</td>
<td>2012-09-23T00:00:00.000Z</td>
</tr>
</tbody>
</table>

**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](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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_TZ(TO_DATETIME('2012-07-05T16:00:00.000Z'), 'America/New_York')</td>
<td>2012-07-05T12:00:00.000Z</td>
</tr>
<tr>
<td>TO_TZ(TO_DATETIME('2012-01-05T16:00:00.000Z'), 'America/New_York')</td>
<td>2012-01-05T11:00:00.000Z</td>
</tr>
<tr>
<td>FROM_TZ(TO_DATETIME('2012-07-05T16:00:00.000Z'), 'America/Los_Angeles')</td>
<td>2012-07-05T23:00:00.000Z</td>
</tr>
<tr>
<td>FROM_TZ(TO_DATETIME('2012-01-05T16:00:00.000Z'), 'America/Los_Angeles')</td>
<td>2012-01-06T00:00:00.000Z</td>
</tr>
</tbody>
</table>
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:

\[
\text{<ExtractExpr> ::= EXTRACT(<expr>,<DateTimeUnit>)}
\]

<table>
<thead>
<tr>
<th>Date Time Unit</th>
<th>Range of Returned Values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECOND</td>
<td>(0 - 59)</td>
<td></td>
</tr>
<tr>
<td>MINUTE</td>
<td>(0 - 59)</td>
<td></td>
</tr>
<tr>
<td>HOUR</td>
<td>(0 - 23)</td>
<td></td>
</tr>
<tr>
<td>DAY_OF_WEEK</td>
<td>(1 - 7)</td>
<td></td>
</tr>
<tr>
<td>DAY_OF_MONTH (DATE)</td>
<td>(1 - 31)</td>
<td></td>
</tr>
<tr>
<td>DAY_OF_YEAR</td>
<td>(1 - 366)</td>
<td></td>
</tr>
<tr>
<td>WEEK</td>
<td>(1 - 53)</td>
<td></td>
</tr>
<tr>
<td>MONTH</td>
<td>(1 - 12)</td>
<td></td>
</tr>
<tr>
<td>QUARTER</td>
<td>(1 - 4)</td>
<td></td>
</tr>
<tr>
<td>YEAR</td>
<td>(-9999 - 9999)</td>
<td></td>
</tr>
<tr>
<td>JULIAN_DAY_NUMBER</td>
<td>(0 - 5373484)</td>
<td></td>
</tr>
</tbody>
</table>

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):

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

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

```sql
DEFINE ClaimsWithAge AS
SELECT ARB(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
FROM SaleState
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 ClaimsWithAge
```

### 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:

```sql
<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.

```sql
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
```
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 \texttt{TRUNC} operation on the TimeStamp attribute:

\begin{verbatim}
RETURN Quarters AS
SELECT SUM(Amount) AS Total,
ARB(TRUNC(TimeStamp, QUARTER)) AS Qtr
FROM SaleState
GROUP BY Qtr
\end{verbatim}

\section*{Using arithmetic operations on date and time values}

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

The following are the supported operations:

\begin{itemize}
\item Add or subtract a duration to or from a time or a dateTime to obtain a new time or dateTime.
\item Subtract two times or dateTimes to obtain a duration.
\item Add or subtract two durations to obtain a new duration.
\item Multiply or divide a duration by a double number.
\item Divide a duration by a duration.
\end{itemize}

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

\begin{tabular}{|c|c|}
\hline
\textbf{Expression} & \textbf{Results} \\
\hline
2012-10-05T00:00:00.000Z + P30D & 2012-11-04T00:00:00.000Z \\
2012-10-05T00:00:00.000Z - PT01M & 2012-10-04T23:59:00.000Z \\
23:00:00.000Z + PT02H & 01:00:00.00 \\
20:00:00.000Z - PT02S & 19:59:58.000Z \\
2012-01-01T00:00:00.000Z - 2012-12-31T00:00:00.000Z & -P365DT0H0M0.000S \\
23:15:00.000Z - 20:12:30.500Z & P0DT3H2M29.500S \\
P1500DT0H0M0.000S - P500DT0H0M0.000S & P1000DT0H0M0.000S \\
P1DT0H30M0.500S * 2.5 & P2DT13H15M1.250S \\
P1DT0H30M0.225S / 2 & P0DT12H15M0.112S \\
P5DT12H00M0.000S / P1DT0H0M0.000S & 5.5 \\
\hline
\end{tabular}
String functions

EQL supports the following string functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT</td>
<td>Concatenates two or more string arguments into a single string.</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>Returns a part (substring) of a character expression.</td>
</tr>
<tr>
<td>TO_STRING</td>
<td>Converts a value to a string.</td>
</tr>
</tbody>
</table>

CONCAT function

CONCAT is a row function that returns a string that is the result of concatenating two or more string values. Its syntax is:

```
CONCAT(string1, string2 [, stringN])
```

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 three literal strings for the arguments:

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

This similar query uses two string-type attributes, plus a quoted space to separate the customer’s first and last names:

```
RETURN results AS
SELECT
  ARB(CONCAT(CUST_FIRST_NAME, ' ', CUST_LAST_NAME)) AS CustomerName
FROM EmployeeState
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 (see below for details).
- `length` is a number that specifies the length of the substring that is to be extracted. If `length` is omitted, EQL returns all characters to the end of `string`. If `length` is less than 1, EQL returns NULL.
Expressions

The *position* argument is treated as follows:

- If *position* is 0, it is treated as 1.
- If *position* is positive, then it is counted from the beginning of *string* to find the first character.
- If *position* is negative, the EQL counts backward from the end of *string*.
- If *position* is greater than the length of *string*, EQL returns NULL.

Note that *position* is not zero indexed. For example, in order to start with the fifth character, *position* must be 5.

**TO_STRING function**

The **TO_STRING** function takes an integer value and returns a string equivalent. Its syntax is:

```
TO_STRING(int)
```

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
  ARB(TO_STRING(P_SIZE)) AS Sizes
FROM ProductState
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 49*.

**Boolean operators**

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

The results of Boolean operations (including the presence of NULL) is shown in the following tables:

**Results of NOT operations:**

<table>
<thead>
<tr>
<th>Value of x</th>
<th>Result of NOT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
### Results of AND operations:

<table>
<thead>
<tr>
<th>Value of x</th>
<th>Value of y</th>
<th>Result of x AND y</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

### Results of OR operations:

<table>
<thead>
<tr>
<th>Value of x</th>
<th>Value of y</th>
<th>x OR y</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>NULL</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>FALSE</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

For information on order of operations, see [Operator precedence rules on page 49](#).
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, such as `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 Dgraph. Note that no conversion is necessary for `mdex:string` attributes.

To determine which conversion function to use, EQL results are accompanied by attribute metadata that describes the type of the attribute.

Using **LOOKUP** expressions for inter-statement references

In EQL, you can define statements and then refer to these statements from other statements via a **LOOKUP** expression.

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 a lookup table to create these types of queries.
Expressions

Syntax for LOOKUP expressions

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

The syntax for a LOOKUP expression is:

```plaintext
<LookupExpr> ::= <statement-name>[<LookupList>].<attribute-name>
```

The square bracket operators are literal and are used to identify the record set and grouping attribute, while the dot operator is also literal and is used to identify the field.

The BNF for LookupList is

```plaintext
<LookupList> ::= <empty>
::= <SimpleExpr> [, <LookupList>]
```

In this BNF syntax, the square brackets indicate the optional use of a second LookupList.

The lookup list corresponds to the grouping attributes of the specified statement. The result is NULL if the lookup 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:

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

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

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

Using LOOKUP against states

LOOKUP expressions are supported where the target statement is referring to a named state, with the rule that there must be exactly one expression inside the square brackets, which is matched against the target state's primary key.

If you use multiple lookup keys against a state, EQL will return an error message similar to this example that uses two lookup keys:

```plaintext
In the definition of attribute "x": The LOOKUP expression has 2 lookup value(s)
; a LOOKUP expression that refers
to state "Sales" must have exactly one lookup value, corresponding to the state's primary key "SalesID"
```
Expressions

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

ARB

ARB selects an arbitrary but consistent value from the set of values in a field.

The syntax of the ARB function is:

```
ARB(<attribute>)
```

where attribute is a single-assign attribute or a set (multi-assign attribute).
Expressions

ARB works as follows:

- For a single-assign attribute, \textit{ARB} first discards all NULL values and then selects an arbitrary but consistent value from the remaining non-NULL values. If the attribute has no non-NULL values, then NULL is returned.
- For a multi-assign attribute, \textit{ARB} looks at all of the rows in the group (including those with empty sets) and selects the set value from one of the rows. In other words, empty sets and non-empty sets are treated equally. This means that because the selection is arbitrary, the returned set value could be an empty set. The \textit{ARB} return type is the same as its argument type: if attribute \( x \) is an \texttt{mdex:long-set}, then so is \( \text{ARB}(x) \). If the attribute has no non-NULL values, then the empty set is returned.

\textbf{ARB examples}

**Single-assign Example:** Price is a single-assign attribute:

\begin{verbatim}
RETURN results AS
SELECT ARB(Price) AS prices
FROM WineState
GROUP BY WineType
ORDER BY WineType
\end{verbatim}

The result for this example is:

<table>
<thead>
<tr>
<th>WineType</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanc de Noirs</td>
<td>16.99</td>
</tr>
<tr>
<td>Bordeaux:</td>
<td>21.99</td>
</tr>
<tr>
<td>Brut</td>
<td>22.99</td>
</tr>
<tr>
<td>Chardonnay:</td>
<td>17.95</td>
</tr>
<tr>
<td>Merlot:</td>
<td>25.99</td>
</tr>
<tr>
<td>Pinot Noir:</td>
<td>14.99</td>
</tr>
<tr>
<td>Red:</td>
<td>9.99</td>
</tr>
<tr>
<td>White:</td>
<td>20.99</td>
</tr>
<tr>
<td>Zinfandel:</td>
<td></td>
</tr>
</tbody>
</table>

Some of the interesting result values from this data set are:

- There are three Bordeaux records: one has a Price assignment of 21.99 and the other two have no Price assignments. Therefore, for the Bordeaux value, \textit{ARB} discarded the two NULL values and returned the 21.99 value.
- There is one Zinfandel record and it does not have a Price assignment. Therefore, a NULL value is returned.

**Multi-assign Example:** Body is a multi-assign attribute:

\begin{verbatim}
RETURN results AS
SELECT ARB(Body) AS bodies
FROM WineState
GROUP BY WineType
ORDER BY WineType
\end{verbatim}

The result for this example is:

<table>
<thead>
<tr>
<th>WineType</th>
<th>bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanc de Noirs</td>
<td>{ Firm, Robust }</td>
</tr>
<tr>
<td>Bordeaux:</td>
<td>{ Silky, Tannins }</td>
</tr>
<tr>
<td>Brut</td>
<td>{ Robust }</td>
</tr>
<tr>
<td>Chardonnay:</td>
<td>{ }</td>
</tr>
<tr>
<td>Merlot:</td>
<td>{ }</td>
</tr>
<tr>
<td>Pinot Noir:</td>
<td>{ Supple }</td>
</tr>
</tbody>
</table>
Some interesting results from this attribute are:

- All nine Red records have at least one Body assignment. The returned value for Red is the \{Silky, Tannins\} set, but, because it is arbitrary, the value could have been any of the other eight sets.
- Two of the White records have Body assignments (and therefore have non-empty sets) while the other two records have no Body assignments (and therefore have empty sets). One of the White empty sets was returned as the arbitrary value, but it just as well could have been one of the non-empty sets.
- Neither of the two Chardonnay records have Body assignments, and therefore the empty set was returned for this group.

**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:

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

where `<attribute>` is the single-assign 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:

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

**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:

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

Oracle® Big Data Discovery: EQL Reference

Version 1.0.0 • January 2015
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, if it appears, must appear at the end of the CASE statement and is equivalent to WHEN TRUE THEN. If no condition matches, the result is NULL or the empty set, depending on the data type of the THEN expressions.

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
FROM SaleState
GROUP BY Class
```

**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 exception integers with doubles (in this case, integers are promoted to doubles).
- COALESCE does not support multi-assign 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 ...
```

## CORRELATION

CORRELATION computes the correlation coefficient between two numeric attributes for all rows within a group. The syntax of the CORRELATION function is:

```
CORRELATION(<arg1>, <arg2>)
```

where each argument is an arbitrary expression or a single-assign numeric (integer or double) attribute. Integer inputs are first promoted to doubles. Note that CORRELATION is symmetric (that is, the same result is returned regardless of which attribute is specified first).

CORRELATION ignores rows in which either argument is NULL and computes the correlation coefficient of the remaining rows. If all rows in the group are NULL, then CORRELATION returns NULL.

The resulting Pearson product-moment correlation coefficient will be a value between +1 and −1 inclusive, where 1 is total positive correlation, 0 is no correlation, and −1 is total negative correlation. Note that there are cases where the output will be NaN (a common case is when there is only a single data point).

### CORRELATION example

In this simple example, WineRating is a single-assign integer attribute while Price is a single-assign double attribute:

```
RETURN results AS
SELECT CORRELATION(WineRating, Price) AS corr
FROM WineState
GROUP
```

The result might be a value of 0.886357407416268.

## HAS_REFINEMENTS

HAS_REFINEMENTS computes whether a particular attribute has non-implicit refinements in the current navigation state.

The syntax of the HAS_REFINEMENTS function is:

```
HAS_REFINEMENTS(<attribute>)
```

where `attribute` is an Endeca attribute (of any data type) in a collection.

In particular, HAS_REFINEMENTS determines if a specific attribute has the same value for every record in a group. If so, then the attribute should be able to return actual refinement values.

### Return values

If `attribute` is an atomic (single-assign) type, the HAS_REFINEMENTS return behavior is:

- If `attribute` is NULL for all rows in the group, HAS_REFINEMENTS returns NULL.
• If attribute the same (non-NULL) value for all rows in the group, \texttt{HAS\_REFINEMENTS} returns FALSE.

• Otherwise (attribute is a mix of values, possibly but not necessarily including NULLs), \texttt{HAS\_REFINEMENTS} returns TRUE.

If attribute is a set (multi-assign) type, the \texttt{HAS\_REFINEMENTS} return behavior is:

• If attribute is the same set for all rows in the group, \texttt{HAS\_REFINEMENTS} returns FALSE. This includes the in which attribute is the empty set for all rows in the group.

• Otherwise, \texttt{HAS\_REFINEMENTS} returns TRUE.

Note that although \texttt{HAS\_REFINEMENTS} tells you if a particular attribute has non-implicit refinements, it does not tell you what they actually are nor does it actually return the refinement values.

\textbf{HAS\_REFINEMENTS example}

In this example, \texttt{HAS\_REFINEMENTS} is used to determine whether the ACCT\_FIRM attribute has available refinements:

\begin{verbatim}
RETURN Result AS
SELECT
  HAS\_REFINEMENTS(ACCT\_FIRM) AS Refs
FROM CorpData
GROUP
\end{verbatim}

In this case, the query returns true, which means that the attribute has available non-implicit refinements.

\textbf{Treating empty sets like NULLs}

If you want \texttt{HAS\_REFINEMENTS} to treat empty sets like NULLs, then you can add a per-aggregate WHERE clause to the \texttt{HAS\_REFINEMENTS} aggregator, as in this example (x is a multi-assign attribute):

\begin{verbatim}
RETURN Result AS
SELECT
  HAS\_REFINEMENTS(x) WHERE (x IS NOT EMPTY) AS Refs
FROM CorpData
GROUP
\end{verbatim}

With this syntax, the \texttt{HAS\_REFINEMENTS} return behavior is:

• If x is the empty set for every row in the group, then \texttt{HAS\_REFINEMENTS(x) WHERE (x IS NOT EMPTY)} is NULL.

• If x is the same non-empty set value for every row in the group, then \texttt{HAS\_REFINEMENTS(x) WHERE (x IS NOT EMPTY)} is FALSE.

• Otherwise, x is a mix of different sets (possibly but not necessarily including the empty set), and \texttt{HAS\_REFINEMENTS(x) WHERE (x IS NOT EMPTY)} is TRUE.

For more information on per-aggregate WHERE clauses, see \textit{Per-aggregation filters on page 46}.

\textbf{IN}

\textbf{IN} expressions perform a membership test.

\textbf{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 \texttt{HAVING} and \texttt{PAGE} expressions.
IN expressions are supported where the target statement is referring to a named state, with the rule that there must be exactly one expression inside the square brackets, which is matched against the target state’s primary key. Note that the expression will be evaluated against the filtered record set of the state, not the unfiltered one.

**IN syntax**

The syntax where the target is a statement is as follows:

```plaintext
[expr1, expr2, ...] IN StatementName
```

The syntax where the target is a state is as follows:

```plaintext
[expr] IN StatementName
```

The reason for this state syntax is that states, like the collections they filter, always have a single key attribute, and thus IN expressions that refer to them must have exactly one expression inside the square brackets.

Note that sets are supported by IN expressions. If one of the named statement’s group keys is a set, then the corresponding expression in the square brackets must be a set of the same type.

**IN example**

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?"

```plaintext
DEFINE HighValueCust AS SELECT
  SUM(SalesAmount) AS Value
FROM SaleState
GROUP BY CustId
HAVING Value > 10000;
RETURN Top_HVC_Products AS SELECT
  COUNT(1) AS NumSales
FROM SaleState
WHERE [CustId] IN HighValueCust
GROUP BY ProductName
ORDER BY NumSales DESC
```

**PERCENTILE**

PERCENTILE computes a specified percentile of the values of an attribute for all records in the group.

The syntax of the PERCENTILE function is:

```plaintext
PERCENTILE(<attribute>, <numeric_literal>)
```

where:

- **attribute** is a single-assign, numeric attribute. The EQL data type for the attribute must be either `mdex:long` or `mdex:double`.

- **numeric_literal** is the percentile to compute. The value must range between 0 (greater than or equal to 0) and 100 (less than or equal to 100). You can specify the value as an integer (such as 50) or a double (such as 50.5). For example, 75 will compute the 75th percentile of an expression. Note that a percentile of 50 is identical to the median.
Expressions

Note that if the percentile falls between two values, then EQL computes a weighted average. As an example, suppose there are only two values, 10 and 20. If you ask for the 20th percentile, then the result will be 12, because 12 is 20% of the way from 10 to 20.

PERCENTILE ignores rows in which its first argument is NULL. If the first argument is NULL for all rows in a group, PERCENTILE returns NULL for that group.

**PERCENTILE examples**

In both examples, SalesAmount is a single-assign double attribute.

This example returns the 90th percentile of the SalesAmount values within the group:

```
RETURN Results AS
SELECT PERCENTILE(SalesAmount, 90) AS x90
FROM SalesState
GROUP
```

The result for this example might be:

```
x90
--------
| 571.18 |
--------
```

This example returns the 25th, 50th, and 75th percentiles of the SalesAmount values within the group:

```
RETURN Results AS
SELECT
    PERCENTILE(SalesAmount, 25) AS x25,
    PERCENTILE(SalesAmount, 50) AS x50,
    PERCENTILE(SalesAmount, 75) AS x75
FROM SalesState
GROUP
```

The result for this example might be:

```
x25   x50   x75
------------------
| 180.225 | 236.5 | 445.675 |
------------------
```

**RECORD_IN_FAST_SAMPLE**

RECORD_IN_FAST_SAMPLE is a row function that returns a Boolean indicating whether the current record is in the sample of the records in the named state.

The syntax of the RECORD_IN_FAST_SAMPLE function is:

```
RECORD_IN_FAST_SAMPLE(<double_literal>)
```

where `double_literal` specifies the size of the requested sample, expressed as a fraction of the total number of records. The sample size must be between 0.0 and 1.0 (inclusive). For example, a value of 0.1 would return approximately 10% of the records in the state.

RECORD_IN_FAST_SAMPLE is intended to be a fast and convenient function for reducing the size of data sent from the Dgraph to Studio (for example, when generating approximate visualizations like heat maps). However, the function does not compute a truly random sample. That is, it is not the case that each record in the collection has the same probability of being chosen, and it is not the case that each subset of k records has the same probability of being chosen as every other subset of k records.
Restrictions on function use

The restrictions for using the `RECORD_IN_FAST_SAMPLE` function are:

- It may appear only as a per-statement `WHERE` condition.
- It may not appear inside a `CASE` expression or as an argument to another function.
- It is allowed only in statements that are `FROM` a single state. EQL will signal an error if `RECORD_IN_FAST_SAMPLE` occurs in a statement `FROM` another statement, `FROM` a view, or `FROM` a `JOIN` or `CROSS`.

Any violation of these restrictions will result in an EQL checking error.

This simple example illustrates the use of the function with the `WHERE` clause:

```sql
RETURN Results AS
SELECT TotalSales AS Sales
FROM SalesState
WHERE RECORD_IN_FAST_SAMPLE(0.1)
```

`RECORD_IN_FAST_SAMPLE` may be used with any of the Boolean operators, as in this similar query:

```sql
RETURN Results AS
SELECT TotalSales AS Sales
FROM SalesState
WHERE TotalSales IS NOT NULL AND RECORD_IN_FAST_SAMPLE(0.1)
```

Note on sampling and joins

Although you may not sample the results of a join (see the third restriction above), you may join the results of sampling. However, be aware that you may not get the desired results. For example, consider this query:

```sql
DEFINE s1 AS
  SELECT ...
  FROM State1
  WHERE RECORD_IN_FAST_SAMPLE(0.1);
DEFINE s2 AS
  SELECT ...
  FROM State2
  WHERE RECORD_IN_FAST_SAMPLE(0.1);
RETURN s3 AS
  SELECT ..
  FROM s1 JOIN s2 ON (...)
```

The results of `s1` and `s2` contain roughly 10% of the records from `State1` and `State2`, respectively. However, in general, the results of `s3` will contain far fewer than 10% of the records it would have, had the previous statements not been sampled.
EQL supports sets, in particular the use of sets to represent multi-assign attributes.

About sets
Aggregate functions
Row functions
Set constructor
Quantifiers
Grouping by sets

About sets

EQL represents multi-assign attributes from collections as sets.

A set consists of a group of elements, typically derived from the values of a multi-assign attribute. EQL sets are intended to behave like mathematical sets: the order of the elements within a set is not specified (and, in general, not observable). An empty set is a set that contains no elements.

All elements in a set must be of the same data type. If the elements in the set come from two multi-assign attributes (for example, by using the INTERSECTION row function), then those two multi-assign attributes must be of the same data type. Sets may not contain duplicate values and sets may not contain other sets.

Sets are constructed in an EQL statement as follows:

- From a reference to a multi-assign attribute. For example, using SELECT with a multi-assign attribute will return the values of that attribute in a set.
- From a single-assign attribute, as an argument to the SET function.
- From an expression that results in a set. For example, using a UNION function will return a set that is a union of two input sets. Note that these set expressions require at least one set on which to operate.
- From a set constructor.

All of these methods are described in this section.

Note that sets are not persistent from one EQL query to another.

Set data types

The data types for sets are:

- mdex:boolean-set for multi-assign Boolean attributes
- mdex:dateTime-set for multi-assign dateTime attributes
• **mdex:double-set** for multi-assign double attributes
• **mdex:duration-set** for multi-assign duration attributes
• **mdex:geocode-set** for multi-assign geocode attributes
• **mdex:long-set** for multi-assign 32-bit integer and 64-bit long attributes
• **mdex:string-set** for multi-assign string attributes
• **mdex:time-set** for multi-assign time attributes

Sets are strictly typed. All of the elements of a specific set must have the same data type. For example, this set:

```
{3, 4.0, 'five'}
```

is invalid because it contains an integer, a double, and a string.

**Sets and NULL**

Sets may not contain NULL values. In addition, sets may not be NULL, but they may be empty. These requirements apply to both multi-assign collection attributes and other expressions of set type.

If a collection record has no assignments for a multi-assign attribute, then in an EQL query, that attribute's value for that record is the empty set.

The results of an EQL statement (whether `DEFINE` or `RETURN`) may contain sets. This means, for instance, that you can define an entity (view) that provides all of the values of a multi-assign attribute to queries that use that entity.

Note that the `IS NULL` and `IS NOT NULL` operations are not supported on sets. Instead, use the `IS_EMPTY` and `IS_NOT_EMPTY` functions to determine whether a set is empty. Likewise, the `IS_EMPTY` and `IS_NOT_EMPTY` functions cannot be used on atomic values (such as on a single-assign attribute).

**Set equality**

Set equality is the same as mathematical set equality: two sets are equal if and only if they contain exactly the same elements, no more, no less. The order of the elements in the set is immaterial. Two empty sets are equal.

Set equality and inequality are defined only on two sets of the same type. For example, you cannot compare an `mdex:long-set` and an `mdex:geocode-set` for equality; doing so will result in an EQL type error.

You can use the `=` (equal) and `<>` (not equal) operators to test for equality between sets. Note that the `<` (less than) and `>` (greater than) operators are not defined for sets.

**Sets, functions, and operators**

This chapter documents the aggregation and row functions that are used with sets.

In addition, sets can be used with the following functions that work on both sets and single-assign attributes, and are documented elsewhere in this guide:

• **`ARB`** on sets looks at all of the rows (both empty sets and non-empty sets) in the group and selects the set value from one of the rows. For details on this function, see `ARB on page 74`.
• **`COUNT`** counts all non-NULL sets (that is, all the sets in the group, including the empty ones). For details, see `COUNT function on page 41`.

Oracle® Big Data Discovery: EQL Reference

Version 1.0.0 • January 2015
• **COUNT_APPROX** also counts all non-NULL sets. For details, see [COUNT_APPROX on page 43](#).
• **COUNTDISTINCT** counts all of the sets, including the empty ones. For details, see [COUNTDISTINCT function on page 43](#).
• **APPROXCOUNTDISTINCT** also counts all of the sets, including the empty ones. For details, see [APPROXCOUNTDISTINCT function on page 44](#).
• **HAS_REFINEMENTS** whether a particular attribute has non-implicit refinements in the current navigation state. For details, see [HAS_REFINEMENTS on page 78](#).

As mentioned above, you can use the = (equal) and <> (not equal) operators to test for equality between sets. The other operators (such as the * multiplication operator) cannot be used on sets.

### Aggregate functions

EQL provides three aggregators for working with sets.
The set aggregate functions can be used only in **SELECT** clauses.

- **SET function**
- **SET_INTERSECTIONS function**
- **SET_UNIONS function**

### SET function

The **SET** aggregation function takes a single-assign attribute and constructs a set of all of the (non-NULL) values from that attribute.

Single-assign attributes have non-set data types (such as `mdex:long`). So the **SET** function takes a non-set data type attribute and produces a set data type result (for example, `mdex:long-set`).

The **SET** function's behavior is as follows:

- All NULL values are discarded. This means that if there are two non-NULL values for an attribute and one NULL value, then only the two non-NULL values are returned.
- If an attribute has no non-NULL values, then the empty set is returned.
- Duplicate values in an attribute are discarded. For example, if three records all have a `WineType=Red` assignment and two of them have `Price=14.95` assignments (the third having `Price=21.95`), then only two Price values (one 14.95 and one 21.95) will be returned for the Red set.
- String values are case-sensitive. Therefore, the string value "Merlot" is distinct from the string value "merlot", which means that they are not duplicate values.
- The order of the values within a set is unspecified and unobservable.

The resulting set will have a set data type (such as `mdex:double-set`). All subsequent operations on it must follow the rules for sets.

The **SET** function is available in one-argument and two-argument versions, as described below. This function can be used only in **SELECT** clauses.
**SET one-argument version**

The syntax of the one-argument version of the `SET` function is:

```
SET(<single-assign_attribute>)
```

where the data type of the attribute must be a non-set data type (such as `mdex:double` for a single-assign double attribute).

In this example, `Price` is a single-assign double attribute:

```
RETURN results AS
SELECT
    SET(Price) AS prices
FROM WineState
GROUP BY WineType
ORDER BY WineType
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>WineType</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanc de Noirs</td>
<td>{ 16.99 }</td>
</tr>
<tr>
<td>Bordeaux</td>
<td>{ 21.99 }</td>
</tr>
<tr>
<td>Brut</td>
<td>{ 22.99, 23.99 }</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>{ 17.95, 34.95 }</td>
</tr>
<tr>
<td>Merlot</td>
<td>{ 25.99 }</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>{ 14.99 }</td>
</tr>
<tr>
<td>White</td>
<td>{ 20.99, 32.99, 43.99 }</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>{}</td>
</tr>
</tbody>
</table>

In the results, note that Zinfandel has an empty set because Zinfandel does not have a Price attribute assignment.

**SET two-argument version**

For situations where the result of the `SET` aggregator can be extremely large (causing the Dgraph to consume excessive memory), a two-argument form of the aggregator is provided to limit the set size.

The syntax of the two-argument version of the `SET` function is:

```
SET(<single-assign_attribute>, <max-size>)
```

where:

- `single-assign_attribute` is an attribute whose data type is a non-set data type (such as `mdex:string` for a single-assign string attribute).
- `max-size` is an integer that specifies the maximum size of the set. If `max-size` is less than the number of elements in the set, The Dgraph arbitrarily chooses which elements to discard; this choice is stable across multiple executions of the query. If `max-size` is 0 (zero) or a negative number, `SET` always returns the empty set.

Note that `max-size` must be an integer literal:

```
SET(Price, 3) is valid.
SET(Price, x) is not valid, even if x is an integer.
```

This sample query is the same as the one-argument example, except that the query limits the sets to a maximum of two elements:

```
RETURN results AS
```
SELECT
  SET(Price, 2) AS prices
FROM WineState
GROUP BY WineType
ORDER BY WineType

The result of this statement might be:

<table>
<thead>
<tr>
<th>WineType</th>
<th>prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blanc de Noirs</td>
<td>{ 16.99 }</td>
</tr>
<tr>
<td>Bordeaux</td>
<td>{ 21.99 }</td>
</tr>
<tr>
<td>Brut</td>
<td>{ 22.99, 23.99 }</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>{ 17.95, 34.95 }</td>
</tr>
<tr>
<td>Merlot</td>
<td>{ 25.99 }</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>{ 14.99 }</td>
</tr>
<tr>
<td>Red</td>
<td>{ 12.99, 9.99 }</td>
</tr>
<tr>
<td>White</td>
<td>{ 20.99, 32.99 }</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>{ }</td>
</tr>
</tbody>
</table>

In the results, note that Red set now has two elements, while it had six elements with the one-argument SET version. Likewise with the White set, which previously had three elements.

Data type errors
When working with the SET function, keep in mind that its resulting sets are of the set data types, such as a mdex:double-set data type.

For example, assume that Price is a multi-assign double attribute. This incorrect example:

RETURN results AS
SELECT SET(Price) AS prices
FROM WineState
GROUP BY WineType
HAVING prices > 10

will throw this error:

In statement "results": In HAVING clause: Cannot compare mdex:double-set and mdex:long

The reason for the error is that the "prices" set is of type mdex:double-set and it is being compared to the number 10 (which is an mdex:double type).

The query should therefore be corrected to something like this:

RETURN results AS
SELECT SET(Price) AS prices
FROM WineState
GROUP BY WineType
HAVING SOME x IN prices SATISFIES (x > 10)

In this example, the SATISFIES expression allows you to make a numerical comparison.

SET_INTERSECTIONS function
The SET_INTERSECTIONS aggregation function takes a multi-assign attribute and constructs a set that is the intersection of all of the values from that attribute.

The syntax of the SET_INTERSECTIONS function is:

SET_INTERSECTIONS(<multi-assign_attribute>)
where the data type of the attribute must be a set data type (such as `mdex:string-set` for a multi-assign string attribute).

This function can be used only in `SELECT` clauses.

**SET_INTERSECTIONS example**

In this example, Body is a multi-assign string attribute:

```
RETURN results AS
SELECT SET_INTERSECTIONS(Body) AS bodyIntersection
FROM WineState
GROUP BY WineType
ORDER BY WineType
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>WineType</th>
<th>bodyIntersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bordeaux</td>
<td>{ Silky, Tannins }</td>
</tr>
<tr>
<td>Brut</td>
<td>{ Robust }</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>{ }</td>
</tr>
<tr>
<td>Merlot</td>
<td>{ }</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>{ Supple }</td>
</tr>
<tr>
<td>Red</td>
<td>{ }</td>
</tr>
<tr>
<td>White</td>
<td>{ }</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>{ Robust, Tannins }</td>
</tr>
</tbody>
</table>

The sets are derived as follows:

- **Bordeaux**: Assigned on three records, with each record having two Body assignments of "Silky" and "Tannins". Therefore, there is an intersection among the three records and a two-element set is returned.
- **Brut**: Assigned on two records, with each record having one Body assignment of "Robust". Therefore, there is an intersection between the two records and a one-element set is returned.
- **Chardonnay**: Assigned on two records, but neither record has a Body assignment. Therefore, there is no intersection between the two records (because there are no values to compare) and the empty set is returned.
- **Merlot**: Assigned on two records, with one record having one Body assignment of "Fruity" and the other record having no Body assignment. Therefore, there is no intersection between the two records and the empty set is returned.
- **Pinot Noir**: Assigned on only one record, which has one Body assignment of "Supple". Therefore, there is an intersection on that record.
- **Red**: Assigned on eight records, with six records having two Body assignments of "Silky" and "Tannins", one record with two Body assignments of "Robust" and "Tannins", and the eighth record with one Body assignment of "Robust". Therefore, there is no intersection among the eight records and the empty set is returned.
- **White**: Assigned on four records, with the first record having two Body assignments of "Fresh" and "Robust", the second record with two Body assignments of "Firm" and "Robust", and the third and fourth records with no Body assignments. Therefore, there is no intersection among the four records and the empty set is returned.
- **Zinfandel**: Assigned on only one record with two Body assignments of "Robust" and "Tannins". Therefore, there is an intersection on that record and a two-element set is returned.
**SET_UNIONS function**

The `SET_UNIONS` aggregation function takes a multi-assign attribute and constructs a set that is the union of all of the values from that attribute.

The syntax of the `SET_UNIONS` function is:

```
SET_UNIONS(<multi-assign_attribute>)
```

where the data type of the attribute must be a set data type (such as `mdex:string-set` for a multi-assign string attribute).

This function can be used only in `SELECT` clauses.

**SET_UNIONS example**

In this example, `Body` is a multi-assign string attribute:

```
RETURN results AS
SELECT SET_UNIONS(Body) AS bodyUnion
FROM WineState
GROUP BY WineType
ORDER BY WineType
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>WineType</th>
<th>bodyUnion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bordeaux</td>
<td>{ Silky, Tannins }</td>
</tr>
<tr>
<td>Brut</td>
<td>{ Robust }</td>
</tr>
<tr>
<td>Chardonnay</td>
<td>[ ]</td>
</tr>
<tr>
<td>Merlot</td>
<td>[ Fruity ]</td>
</tr>
<tr>
<td>Pinot Noir</td>
<td>{ Supple }</td>
</tr>
<tr>
<td>Red</td>
<td>{ Robust, Silky, Tannins }</td>
</tr>
<tr>
<td>White</td>
<td>{ Firm, Fresh, Robust }</td>
</tr>
<tr>
<td>Zinfandel</td>
<td>{ Robust, Tannins }</td>
</tr>
</tbody>
</table>

The sets are derived as follows:

- **Bordeaux**: Assigned on three records, with each record having two `Body` assignments of “Silky” and “Tannins”. Therefore, the union returns a two-element set of the two assignments.
- **Brut**: Assigned on two records, with each record having one `Body` assignment of "Robust". Therefore, the union returns a one-element set with "Robust".
- **Chardonnay**: Assigned on two records, but neither record has a `Body` assignment. Therefore, the union is empty.
- **Merlot**: Assigned on two records, with one record having one `Body` assignment of "Fruity" and the other record having no `Body` assignment. Therefore, there is a union of the single assignment on the one record.
- **Pinot Noir**: Assigned on only one record, which has one `Body` assignment of "Supple". Therefore, there is a union on that record.
- **Red**: Assigned on eight records, with six records having two `Body` assignments of "Silky" and "Tannins", one record with two `Body` assignments of "Robust" and "Tannins", and the eighth record with one `Body` assignment of "Robust". Therefore, the resulting union produces a three-element set of the three distinct assignments.
Sets and Multi-assign Data

- White: Assigned on four records, with the first record having two Body assignments of "Fresh" and "Robust", the second record with two Body assignments of "Firm" and "Robust", and the third and fourth records with no Body assignments. Therefore, there is a union of the "Firm", "Fresh", and "Robust" assignments.

- Zinfandel: Assigned on only one record with two Body assignments of "Robust" and "Tannins". Therefore, there is a union on that record.

Row functions

EQL provides a number of row functions for working with sets.

The set row functions can be used anywhere that an arbitrary expression can be used. For example, they can be used in SELECT clauses, WHERE clauses, ORDER BY clauses, and so on.

ADD_ELEMENT function
CARDINALITY function
DIFFERENCE function
FOREACH function
INTERSECTION function
IS EMPTY and IS NOT EMPTY functions
IS MEMBER OF function
SINGLETON function
SUBSET function
TRUNCATE_SET function
UNION function

ADD_ELEMENT function

The ADD_ELEMENT row function adds an element to a set.

ADD_ELEMENT takes an atomic value and a set and returns that set with the atomic value added to it. The atomic value must be of the same data type as the current elements in the set. The atomic value is not added to the set if a duplicate value is already in the set. Note that the atomic value is not added to the set in the Dgraph, but only to the new, temporary set that is created by the ADD_ELEMENT function.

The syntax of the ADD_ELEMENT function is:

```
ADD_ELEMENT(<atomic-value>, <set>)
```

where:

- **atomic-value** is an atomic value, such as 50 for an integer set or 'fifty' for a string set. It can also be a single-assign attribute. **atomic-value** will be added to **set**. The type of the atomic value must match the type of the set's elements.
• *set* is a set to which *atomic-value* will be added. The elements of *set* must have the same set data type as *atomic-value*. For example, if *atomic-value* is a single-assign double attribute, then the elements of *set* must also be strings.

Examples of some results are as follows (*{}* indicates an empty set):

- `ADD_ELEMENT(1, { 2, 3 }) = { 1, 2, 3 }`
- `ADD_ELEMENT(1, { 1, 2 }) = { 1, 2 }`
- `ADD_ELEMENT(NULL, { 1, 2 }) = { 1, 2 }`
- `ADD_ELEMENT(1, { 'a', 'b' })` yields a checking error because the atomic value and the set elements are not of the same data type

### ADD_ELEMENT examples

**Example 1:** In this example, the number 100 is added to the Score integer set (which currently does not have a value of 100 in it):

```
RETURN results AS
SELECT
    WineID AS idRec,
    ADD_ELEMENT(100, Score) AS addAttrs
FROM WineState
WHERE WineID BETWEEN 10 AND 14
ORDER BY idRec
```

The result of this statement might be:

```
addAttrs    idRec
-------------------------
( 100, 83, 85, 86 )   10
( 100, 82, 83 )       11
( 100, 81, 89 )       12
( 100, 73, 75 )       13
( 100, 72, 74, 75 )   14
-------------------------
```

The results show that the number 100 was added to the sets. For example, the Score set of Record 12 previously had 81 and 89 as its elements, but now has 81, 89, and 100 as the element values.

### CARDINALITY function

The **CARDINALITY** row function takes a set and returns the number of elements in that set.

The syntax of the **CARDINALITY** function is:

```
CARDINALITY(<set>)
```

where *set* is a set of any set data type (such as `mdex:string-set` or `mdex:long-set`). For example, *set* can be a multi-assign double attribute.

### CARDINALITY example

In this example, Body is a multi-assign string attribute and WineID is the primary key of the records:

```
RETURN results AS
SELECT
    WineID AS id,
    CARDINALITY(Body) AS numBody
FROM WineState
WHERE WineID < 7
ORDER BY id
```
The result of this statement might be:

<table>
<thead>
<tr>
<th>id</th>
<th>numBody</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

The numBody column shows the number of elements in the Body set for each record.

**DIFFERENCE function**

The **DIFFERENCE** row function takes two sets of the same data type and returns a set containing all of the elements of the first set that do not appear in the second set.

The syntax of the **DIFFERENCE** function is:

```
DIFFERENCE(<set1>, <set2>)
```

where:

- `set1` is a set of any set data type (such as `mdex:string-set`). For example, `set1` can be a multi-assign string attribute.
- `set2` is a set of the same set data type as `set1`. For example, if `set1` is a multi-assign string attribute, then `set2` must also be a set of strings (such as another multi-assign string attribute).

Examples of some results are as follows (`{ }` indicates an empty set):

```
DIFFERENCE({ 1, 2, 3, 4, 5 }, { 1, 3, 5 }) = { 2, 4 }
DIFFERENCE({ }, { 1, 3, 5 }) = { }
DIFFERENCE({ 1, 2, 3 }, { }) = { 1, 2, 3 }
DIFFERENCE({ 1, 2 }, { 'a', 'b' }) yields a checking error because the two sets are not of the same data type
```

**DIFFERENCE example**

In the examples below, both Body and Flavors are multi-assign string attributes. Their values for five records are:

Record 5: Body=Earthy, Silky, Tannins  
    Flavors=Blackberry, Earthy, Toast  
Record 6: Body=Robust  
    Flavors=Berry, Plum, Zesty  
Record 7: Body=Silky, Tannins  
    Flavors=Cherry, Pepper, Prune  
Record 8: Body=Oak, Robust  
    Flavors=Cherry, Oak, Raspberry  
Record 9: Body=Fruit, Strawberry, Silky, Tannins  
    Flavors=Fruit, Earthy, Strawberry

First, we want all the elements of the Body set that do not appear in the Flavors set:

```
RETURN results AS  
SELECT WineID AS idRec,  
    DIFFERENCE(Body, Flavors) AS diffAttrs  
FROM WineState  
WHERE WineID BETWEEN 5 AND 9
```
The result of this statement might be:

<table>
<thead>
<tr>
<th>diffAttrs</th>
<th>idRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Silky, Tannins }</td>
<td>5</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>6</td>
</tr>
<tr>
<td>{ Silky, Tannins }</td>
<td>7</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>8</td>
</tr>
<tr>
<td>{ Silky, Tannins }</td>
<td>9</td>
</tr>
</tbody>
</table>

Records 5, 7, and 9 have "Silky" and "Tannins" in the Body set, but these values do not appear in the Flavors set. Likewise, Records 6 and 8 have "Robust" in the Body set, but that value does not appear in the Flavors set.

We then reverse the difference comparison between the two sets. The statement is identical to the first example, except that Flavors is the first argument rather than Body:

```
RETURN results AS
SELECT
  WineID AS idRec,
  DIFFERENCE(Flavors, Body) AS diffAttrs
FROM WineState
WHERE WineID BETWEEN 5 AND 9
ORDER BY idRec
```

This time, the result of this statement will look different:

<table>
<thead>
<tr>
<th>diffAttrs</th>
<th>idRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Blackberry, Toast }</td>
<td>5</td>
</tr>
<tr>
<td>{ Berry, Plum, Zesty }</td>
<td>6</td>
</tr>
<tr>
<td>{ Cherry, Pepper, Prune }</td>
<td>7</td>
</tr>
<tr>
<td>{ Cherry, Raspberry }</td>
<td>8</td>
</tr>
<tr>
<td>{ Earthy }</td>
<td>9</td>
</tr>
</tbody>
</table>

To take Record 9 as an example of the output, "Earthy" is the only element from the first set (the Flavors set) that does not appear in the second set (the Body set).

**FOREACH function**

The **FOREACH** function performs a computation on every member of a set or sets and assembles the results into a set.

**FOREACH** may be used in any context within an EQL statement that accepts expressions: LET, SELECT, row function or aggregator arguments, WHERE, HAVING, or ORDER BY. Because **FOREACH** always evaluates to a set, the context must accept set-typed expressions, or EQL will signal a checking error.

**Syntax**

The syntax of the **FOREACH** function is:

```
FOREACH id_1 IN set_1[, id_n IN set_n] RETURN(expression)
```

where:

- *id* is an arbitrary identifier for the item to be computed. The identifier must use the NCName format.
- *set* is a set of any set data type.
• expression is an EQL expression. The expression must be enclosed within parentheses and the 
RETURN keyword is required. The function must have only one RETURN regardless of the number of id 
parameters are used.

At a minimum, you must specify one identifier/set pair (id IN set) and the RETURN expression.

Scope and Shadowing

FOREACH binds the id_1 through id_n identifiers within the RETURN expression; they are not in scope in the 
universes set_1 through set_n. These bindings shadow any other bindings for id_1 through id_n that may be 
in scope at the point of the FOREACH function.

As with the EVERY and SOME quantifiers, EQL does not allow references to these bound variables to be 
qualified with data-source aliases. For example, in this statement:

```
RETURN results AS
  SELECT
    Source.x AS x,
    FOREACH x IN {1, 2}, y in {3, 4} RETURN (results.x + Source.y) AS vals
FROM Source
```

EQL interprets the reference to results.x (in the FOREACH RETURN expression) as a reference to the x defined 
by the SELECT clause (that is, as an alias for Source.x and not as a reference to the x bound by FOREACH). 
Similarly, EQL interprets the reference Source.y as a reference to the attribute y in Source.

However, if you drop the statement qualifiers, as in the following:

```
RETURN results AS
  SELECT
    Source.x AS x,
    FOREACH x IN {1, 2}, y in {3, 4} RETURN (x + y) AS vals
FROM Source
```

then EQL interprets x and y (in the FOREACH RETURN expression) as references to the FOREACH-bound 
identifiers, even though x and y are already in scope from the earlier SELECT and from the data source. 
Therefore, vals always has the value {4, 5, 6}.

Types

In the FOREACH syntax, set_1 through set_n must be a set data type (such as mdex:string-set); EQL 
signals an error otherwise. The corresponding x_1 through x_n identifiers must have the type of the elements 
of these sets. To illustrate, consider this example:

```
FOREACH x IN {1, 2}, y IN {'abc', 'def'} RETURN (x + y)
```

Here, the first universe is a set of integers, and thus x has type integer within the RETURN expression. 
Similarly, the second universe is a set of strings, so y has type string. As a result, the RETURN expression x + 
y is thus ill-typed (and EQL signals an error accordingly).

If the RETURN expression has atomic type t, then the entire FOREACH expression has type "set of t". 
Therefore, if the RETURN expression has type integer, then the containing FOREACH expression has type 
integer set.

If, on the other hand, the RETURN expression has a set type, then the FOREACH expression has the same type 
as the RETURN expression. (This corresponds to the case where FOREACH takes the union of the values of the 
RETURN expression.) So, if the RETURN expression produces a set of string, then the FOREACH expression 
does also.
FOREACH and aggregation

FOREACH expressions may appear in both pre-grouping and post-grouping computation (including WHERE, HAVING, and ORDER BY clauses). They interact with aggregations in much the same way that quantifier expressions (EVERY and SOME) do:

- FOREACH expressions can appear inside aggregator arguments:

  ```sql
  RETURN results AS
  SELECT
    SET_UNIONS(FOREACH x IN e RETURN(b)) AS unions
  FROM Source
  GROUP
  ```

- An aggregator must not appear between a FOREACH and its bound variable. That is, the following is invalid, and EQL signals an error accordingly:

  ```sql
  FOREACH x IN e RETURN(SUM(x))
  ```

Operational details

To explain how FOREACH works, we start with a simple example:

```sql
FOREACH x IN {1, 2, 3} RETURN(x * 2 + 1)
```

This expression evaluates to the set {3, 5, 7}. This is because, notionally, EQL evaluates the RETURN expression \( x \times 2 + 1 \) once for each member of the set \( \{1, 2, 3\} \), with \( x \) taking on each element of that set in turn. Finally, EQL assembles the results of these evaluations into another set.

Because the universe and result are both sets, you cannot specify the order in which the traversal visits the elements of the universe, or the order in which the result values appear in the final set.

You can also use FOREACH to iterate over multiple sets:

```sql
FOREACH x IN {1, 2, 3}, y IN {40, 50, 60} RETURN(x + y)
```

This expression evaluates to the set \( \{41, 42, 43, 51, 52, 53, 61, 62, 63\} \). Here, EQL evaluates the RETURN expression \( x + y \) for all possible combinations of values \( x \) and \( y \) from the two sets:

\[
\begin{align*}
  x = 1, y = 40: x + y &= 41 \\
  x = 2, y = 40: x + y &= 42 \\
  x = 3, y = 40: x + y &= 43 \\
  x = 1, y = 50: x + y &= 51 \\
  \ldots \\
  x = 3, y = 60: x + y &= 63
\end{align*}
\]

Because sets cannot contain duplicate values or NULLs, the result of a FOREACH expression may be smaller than the universe (or the cross product of the universes), as illustrated by these two examples:

```sql
FOREACH x IN {-1, 1, 2} RETURN (ABS(x)) // returns \{1, 2\}
FOREACH x IN {'3', '4', 'y'} RETURN (TO_STRING(x)) // returns \{3, 4\}
```

In the first example, \( \text{ABS}(-1) = \text{ABS}(1) = 1 \), so the final set contains only two elements; the value 1 may not appear twice. In the second example, \( \text{TO_INTEGER('x')} \) is NULL, so this value does not appear in the final set.

If the RETURN expression itself produces a set (rather than a single value), then FOREACH evaluates as described above, but computes the union of all of the RETURN sets as the final result. For example:

```sql
FOREACH x in {3, 4, 5}, y IN {-1, 0, 2} RETURN ((x + y, x - y))
```

evaluates to the set \( \{1, 2, 3, 4, 5, 6, 7\} \), as follows:

// Note: "body" is the RETURN expression
Sets and Multi-assign Data

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>body</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-1</td>
<td>{2,4}</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>{3}</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>{1,5}</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>{3,5}</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>{2,6}</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>{4,6}</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>{3,7}</td>
</tr>
</tbody>
</table>

Computing the union of all of the sets in the "body" column produces the final result.

An important corollary of the above description is that if any of the universe sets are empty, then the result set is itself empty. For example, given two multi-assign attributes whose assigned values are:

<table>
<thead>
<tr>
<th>xs</th>
<th>ys</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2}</td>
<td>{3}</td>
</tr>
<tr>
<td>{}</td>
<td>{4,5}</td>
</tr>
<tr>
<td>{6}</td>
<td>{}</td>
</tr>
</tbody>
</table>

then the statement:

```
RETURN results AS
SELECT
  xs,
  ys,
  FOREACH x IN xs, y IN ys RETURN (x + y) AS zs
FROM InputState
```

produces the following results:

<table>
<thead>
<tr>
<th>xs</th>
<th>ys</th>
<th>zs</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2}</td>
<td>{3}</td>
<td>{4,5}</td>
</tr>
<tr>
<td>{}</td>
<td>{4,5}</td>
<td>{}</td>
</tr>
<tr>
<td>{6}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

FOREACH Examples

Example 1: This is one of the simplest examples of a FOREACH expression, as only one multi-assign integer attribute (Score) is used and the RETURN expression just returns the values of each Score set:

```
RETURN Results AS
SELECT
  WineID AS id,
  FOREACH x IN Score RETURN(x) AS ratings
FROM WineState
ORDER BY id
```

Example 2: This example uses two multi-assign string attributes (Body and Flavors) and concatenates the members of the sets:

```
RETURN Results AS
SELECT
  WineID AS id,
  FOREACH x IN Body, y IN Flavors RETURN (CONCAT(x, ' ', y)) AS bodyflavor
FROM WineState
WHERE IS_NOT_EMPTY(Body) AND IS_NOT_EMPTY(Flavors)
ORDER BY id
```
Note that the WHERE clause uses two IS_NOT_EMPTY functions to prevent empty sets being selected.

**Example 3:** This example uses both LET and FOREACH, and also uses the EXTRACT function in the RETURN expression:

```eql
RETURN Results AS
LET
  (FOREACH d IN ShipDate RETURN (EXTRACT(d, YEAR))) AS yearSet
SELECT
  SET(Price) AS prices
FROM WineState
GROUP BY MEMBERS(yearSet) AS shipyear
```

In the example, **ShipDate** is a multi-assign dateTime attribute. The attribute **d** is visible only within the RETURN expression, and it shadows any other attribute by the same name within that expression. Note that, because **yearSet** is defined in a LET clause rather than a SELECT clause, it will not appear in the statement's results.

**INTERSECTION function**

The **INTERSECTION** row function takes two sets of the same data type and returns a set that is the intersection of both input sets.

The syntax of the **INTERSECTION** function is:

```
INTERSECTION(<set1>, <set2>)
```

where:
- **set1** is a set of any set data type (such as mdex:string-set). For example, **set1** can be a multi-assign string attribute.
- **set2** is a set of the same set data type as **set1**. For example, if **set1** is a multi-assign string attribute, then **set2** must also be a set of strings (such as another multi-assign string attribute).

If an attempt is made to intersect two sets of different set data types, an error message is returned similar to this example:

```
The function "INTERSECTION" is defined for the argument type(s) mdex:string-set, mdex:double-set
```

In this error case, **INTERSECTION** was used with a multi-assign string attribute (**mdex:string-set**) and a multi-assign double attribute (**mdex:double-set**) as inputs.

**INTERSECTION example**

In this example, both **Body** and **Flavors** are multi-assign string attributes and **WineID** is the primary key of the records:

```
RETURN results AS
SELECT
  WineID AS idRec,
  INTERSECTION(Body, Flavors) AS intersectAttrs
FROM WineState
WHERE WineID BETWEEN 5 AND 9
ORDER BY idRec
```

The result of this statement might be:

```
idRec  intersectAttrs
--------
| 5     | { Earthy } |
| 6     | { }        |
```
Records 5 and 8 have one-element result sets because there is one intersection between their Body and Flavors assignments, while Record 9 has a two-element intersection. Records 6 and 7 return empty sets because there is no intersection among their Body and Flavors assignments.

**IS_EMPTY and IS_NOT_EMPTY functions**

The **IS EMPTY** and **IS NOT EMPTY** functions determine whether a set is or is not empty. The **IS EMPTY** and **IS NOT EMPTY** functions provide alternative syntaxes for these functions.

⚠️ Note: The **IS NULL** and **IS NOT NULL** operations are not supported on sets.

**Sample data for the examples**

The sample data used to illustrate these functions consists of a Body multi-assign string attribute and five records:

<table>
<thead>
<tr>
<th>Rec ID</th>
<th>Body attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>{ Silky, Tannins }</td>
</tr>
<tr>
<td>17</td>
<td>{}</td>
</tr>
<tr>
<td>18</td>
<td>{ Silky, Tannins }</td>
</tr>
<tr>
<td>19</td>
<td>{ Fresh, Robust }</td>
</tr>
<tr>
<td>20</td>
<td>{}</td>
</tr>
<tr>
<td>21</td>
<td>{}</td>
</tr>
<tr>
<td>22</td>
<td>{ Firm, Robust }</td>
</tr>
</tbody>
</table>

Three of the records have no Body assignment (and therefore are empty sets), while the other three records have two Body assignments.

Note that these functions are used in **WHERE** clauses in the examples. However, they can be used anywhere that an arbitrary expression can be used, such as in **SELECT** and **HAVING** clauses.

**IS_EMPTY function**

The **IS_EMPTY** function takes a set and returns **TRUE** if that set is empty. The syntax of the **ISEMPTY** function is:

```sql
IS_EMPTY(set)
```

where **set** is a set of any set data type (such as `mdex:string-set` or `mdex:long-set`). For example, **set** can be a multi-assign double attribute.

Examples of two results are as follows (note that `{ }` indicates an empty set):

```sql
IS_EMPTY({ }) = TRUE
IS_EMPTY({ 1 }) = FALSE
```

In this example, the Body attribute is checked for emptiness:

```sql
RETURN results AS
SELECT
  WineID AS idRec,
  Body AS bodyAttr
WHERE (WineID BETWEEN 16 AND 22) AND (IS_EMPTY(Body))
```
The result of this statement would be:

<table>
<thead>
<tr>
<th>idRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

In the result, only Records 17, 20, and 21 are returned because they have an empty Body set.

**IS EMPTY function**

The **IS EMPTY** function provides an alternative syntax to **IS_EMPTY** and also returns **TRUE** if that set is empty.

The syntax of the **IS EMPTY** function is:

```sql
<set> IS EMPTY
```

where `set` is a set of any set data type, such as a multi-assign double attribute.

The previous **IS_EMPTY** example can be re-written as follows:

```sql
RETURN results AS
SELECT
    WineID AS idRec,
    Body AS bodyAttr
FROM WineState
WHERE (WineID BETWEEN 16 AND 22) AND (Body IS EMPTY)
ORDER BY idRec
```

The results of this example would the same as the previous **IS_EMPTY** example.

**IS_NOT_EMPTY function**

The **IS_NOT_EMPTY** function takes a set and returns **TRUE** if that set is not empty. The syntax of the **IS_NOT_EMPTY** function is:

```sql
IS_NOT_EMPTY(<set>)
```

where `set` is a set of any set data type. For example, `set` can be a multi-assign geocode attribute.

Examples of two results are as follows (`{ }` indicates an empty set):

```sql
IS_NOT_EMPTY({ }) = FALSE
IS_NOT_EMPTY({ 1 }) = TRUE
```

In this example, the Body attribute is checked for non-emptiness:

```sql
RETURN results AS
SELECT
    WineID AS idRec,
    Body AS bodyAttr
FROM WineState
WHERE (WineID BETWEEN 16 AND 22) AND (IS_NOT_EMPTY(Body))
ORDER BY idRec
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>bodyAttr</th>
<th>idRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Silky, Tannins )</td>
<td>16</td>
</tr>
<tr>
<td>( Silky, Tannins )</td>
<td>18</td>
</tr>
</tbody>
</table>
In the result, Records 16, 18, 19, and 22 are returned because they have non-empty Body sets. However, Records 17, 20, and 21 are not returned because there is no Body assignment for those records (and therefore those sets would be empty).

**IS NOT EMPTY function**

The *IS NOT EMPTY* function provides an alternative syntax to *IS_NOT_EMPTY* and also returns *TRUE* if that set is not empty.

The syntax of the *IS NOT EMPTY* function is:

```
<set> IS NOT EMPTY
```

where *set* is a set of any set data type, such as a multi-assign string attribute.

The previous *IS_NOT_EMPTY* example can be re-written as follows:

```sql
RETURN results AS
SELECT
    WineID AS idRec,
    Body AS bodyAttr
FROM WineState
WHERE (WineID BETWEEN 16 AND 22) AND (Body IS NOT EMPTY)
ORDER BY idRec
```

The results of this example would the same as the previous *IS_NOT_EMPTY* example.

**IS_MEMBER_OF function**

The *IS_MEMBER_OF* row function takes an atomic value and a set, and returns a Boolean indicating whether the atomic value occurs in the set.

The syntax of the *IS_MEMBER_OF* function is:

```
IS_MEMBER_OF(<atomic-value>, <set>)
```

where:

- *atomic-value* is an atomic value, such as 50 (for an integer set) or 'test' (for a string set). It can also be a single-assign attribute. *atomic-value* will be checked to see whether it occurs in *set*. The type of the atomic value must match the type of the set's elements.

- *set* is a set in which its elements have the same set data type as *atomic-value*. For example, if *atomic-value* is a single-assign string attribute, then the elements of *set* must also be strings.

Examples of some results are as follows (*{}* indicates an empty set):

```
IS_MEMBER_OF(1, {}) = FALSE
IS_MEMBER_OF(1, { 1, 2, 3 }) = TRUE
IS_MEMBER_OF(1, { 2, 3, 4 }) = FALSE
IS_MEMBER_OF(NULL, {}) = NULL
IS_MEMBER_OF(NULL, { 1, 2, 3 }) = NULL
IS_MEMBER_OF('a', {'a', 'b', 'c'}) yields a checking error because the atomic value and the set elements are not of the same data type
```

The *IS_MEMBER_OF* function is intended as a membership check function.
### IS_MEMBER_OF examples

**Example 1:** In this example, the statement determines whether the number 82 (which is an integer) occurs in the Score set (which has integer elements):

```
RETURN results AS
SELECT
    WineID AS idRec,
    IS_MEMBER_OF(82, Score) AS memberAttrs
FROM WineState
WHERE WineID BETWEEN 22 AND 25
ORDER BY idRec
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>idRec</th>
<th>memberAttrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>false</td>
</tr>
<tr>
<td>23</td>
<td>true</td>
</tr>
<tr>
<td>24</td>
<td>false</td>
</tr>
<tr>
<td>25</td>
<td>true</td>
</tr>
</tbody>
</table>

The results show that the number 82 occurs in the Score set of Records 23 and 25, but not in Records 22 and 24.

**Example 2:** This example is similar to Example 1, except that it uses the Ranking single-assign integer attribute as the first argument to the `IS_MEMBER_OF` function and the Score set (which has integer elements) as the second argument:

```
RETURN results AS
SELECT
    WineID AS idRec,
    IS_MEMBER_OF(Ranking, Score) AS memberAttrs
FROM WineState
ORDER BY idRec
```

**Example 3:** This example is similar to Example 2, except that it uses the `IS_MEMBER_OF` function in a **WHERE** clause:

```
RETURN results AS
SELECT
    WineID AS idRec,
    Price AS prices
FROM WineState
WHERE IS_MEMBER_OF(Ranking, Score) AND Price IS NOT NULL
ORDER BY idRec
```

### Using the IN expression

You can use the **IN** expression as an alternative to the `IS_MEMBER_OF` function for membership tests. To illustrate this, Example 3 can be re-written as:

```
RETURN results AS
SELECT
    WineID AS idRec,
    Price AS prices
FROM WineState
WHERE Ranking IN Score AND Price IS NOT NULL
ORDER BY idRec
```

For details on the **IN** expression, see **IN on page 79.**
**SINGLETON function**

The **SINGLETON** function takes a single atomic value and returns a set containing only that value.

The syntax of the **SINGLETON** function is:

```
SINGLETON(<atomic-value>)
```

where *atomic-value* is an atomic value, such as 50 for an integer set or 'fifty' for a string set. It can also be a single-assign attribute. The resulting set will contain only *atomic-value*.

Examples of some results are as follows ({} indicates an empty set):

```
SINGLETON(NULL) = {}  
SINGLETON(1) = { 1 }  
SINGLETON('a') = { 'a' }
```

**SINGLETON example**

In this example, WineType is a single-assign string attribute and WineID is the primary key of the records:

```
RETURN results AS
SELECT
    WineID AS idRec,
    SINGLETON(WineType) AS singleAttr
FROM WineState
WHERE WineID BETWEEN 10 AND 14
ORDER BY idRec
```

The result of this statement might be:

```
idRec singleAttr
----------------------
| 10 | { Bordeaux } |
| 11 | { Zinfandel } |
| 12 | { Red } |
| 13 | { Bordeaux } |
| 14 | { Merlot } |
----------------------
```

**SUBSET function**

The **SUBSET** row function takes two sets of the same data type and returns a Boolean true if (and only if) the second set is a subset of the first set.

The syntax of the **SUBSET** function is:

```
SUBSET(<set1>, <set2>)
```

where:

- *set1* is a set of any set data type (such as `mdex:string-set`). For example, *set1* can be a multi-assign string attribute.

- *set2* is a set of the same set data type as *set1*. For example, if *set1* is a multi-assign string attribute, then *set2* must also be a set of strings (such as another multi-assign string attribute). *set2* will be checked to see if it is completely contained within *set1*.

For example, assuming this statement:

```
SUBSET(A, B)
```

then the **SUBSET** result is true if (and only if) B is a subset of A.
Other examples of some results are as follows ({} indicates an empty set):

<table>
<thead>
<tr>
<th>SUBSET(( ), ( )) = TRUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSET((1, 2, 3), ( )) = TRUE</td>
</tr>
<tr>
<td>SUBSET((1, 2), (1, 2)) = TRUE</td>
</tr>
<tr>
<td>SUBSET((1, 2, 3), (1, 2)) = TRUE</td>
</tr>
<tr>
<td>SUBSET((1, 2), (1, 2)) = FALSE</td>
</tr>
<tr>
<td>SUBSET((1, 2), (&quot;x&quot;, &quot;y&quot;, &quot;z&quot;)) yields a checking error because the two sets are not of the same data type</td>
</tr>
</tbody>
</table>

Note that the empty set is always a subset of every other set (including the empty set).

**SUBSET example**

In this example, both Flavors and Body are multi-assign string attributes, and WineID is the primary key of the records:

```
RETURN results AS
SELECT
    WineID AS id,
    SUBSET(Body, Flavors) AS subAttrs
FROM WineState
WHERE WineID < 5
ORDER BY id
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>id</th>
<th>subAttrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
</tr>
</tbody>
</table>

The results show that the Flavors set is a subset of the Body set in Records 1 and 2, but not in Records 3 and 4.

**TRUNCATE_SET function**

The TRUNCATE_SET row function takes a set and an integer, and returns a copy of the set with no more than the specified number of elements in it.

The syntax of the TRUNCATE_SET function is:

```
TRUNCATE_SET(<set>, <max-size>)
```

where:

- `set` is a set of any set data type (such as mdex:string-set or mdex:long-set). For example, `set` can be a multi-assign string attribute.

- `max-size` is an integer that specifies the maximum size of the truncated set. If `max-size` is less than the number of elements in the set, the Dgraph arbitrarily chooses which elements to discard; this choice is stable across multiple executions of the query. If `max-size` is 0 (zero) or a negative number, the empty set is returned.

Examples of some results are as follows ({} indicates an empty set):

<table>
<thead>
<tr>
<th>TRUNCATE_SET(( ), 2) = {}</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUNCATE_SET((&quot;a&quot;, &quot;b&quot;), 2) = {&quot;a&quot;, &quot;b&quot;}</td>
</tr>
<tr>
<td>TRUNCATE_SET((&quot;a&quot;, &quot;b&quot;, &quot;c&quot;), 2) = {&quot;b&quot;, &quot;c&quot;}</td>
</tr>
<tr>
<td>TRUNCATE_SET((1, 2), 20) = {1, 2}</td>
</tr>
</tbody>
</table>
TRUNCATE_SET({ 1, 2 }, -3) = { }

TRUNCATE_SET is useful when you want to ensure that final results of a set are of a reasonable and manageable size for your front-end UI.

**TRUNCATE_SET example**

In this example, Flavors is a multi-assign string attribute and WineID is the primary key of the records:

```
RETURN results AS
SELECT
  WineID AS id,
  Flavors AS fullFlavors,
  TRUNCATE_SET(fullFlavors, 1) AS truncFlavors
FROM WineState
WHERE WineID BETWEEN 15 AND 19
ORDER BY id
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>fullFlavors</th>
<th>id</th>
<th>truncFlavors</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Blackberry, Oaky, Strawberry }</td>
<td>15</td>
<td>{ Blackberry }</td>
</tr>
<tr>
<td>{ Currant, Licorice, Tobacco }</td>
<td>16</td>
<td>{ Licorice }</td>
</tr>
<tr>
<td>{ Cedar, Cherry, Spice }</td>
<td>17</td>
<td>{ Cherry }</td>
</tr>
<tr>
<td>{ Black Cherry, Cedar, Fruit }</td>
<td>18</td>
<td>{ Black Cherry }</td>
</tr>
<tr>
<td>{ Herbal, Strawberry, Vanilla }</td>
<td>19</td>
<td>{ Herbal }</td>
</tr>
</tbody>
</table>

The fullFlavors set shows the full set of Flavors assignments on each of the five chosen records. The fullFlavors set is then truncated to a one-element set.

**UNION function**

The **UNION** row function takes two sets of the same data type and returns a set that is the union of both input sets.

The syntax of the **UNION** function is:

```
UNION(<set1>, <set2>)
```

where:

- **set1** is a set of any set data type (such as `mdex:string-set`). For example, **set1** can be a multi-assign string attribute.

- **set2** is a set of the same set data type as **set1**. For example, if **set1** is a multi-assign string attribute, then **set2** must also be a set of strings (such as another multi-assign string attribute).

If an attempt is made to union two sets of different set data types, an error message is returned similar to this example:

```
The function "UNION" is not defined for the argument type(s) mdex:string-set, mdex:double-set
```

In this error case, **UNION** was used with a multi-assign string attribute (`mdex:string-set`) and a multi-assign double attribute (`mdex:double-set`) as inputs.
UNION example

In this example, both Body and Flavors are multi-assign string attributes and WineID is the primary key of the records:

```
RETURN results AS
SELECT
    WineID AS idRec,
    UNION(Body, Flavors) AS unionAttrs
FROM WineState
WHERE WineID BETWEEN 5 AND 9
ORDER BY idRec
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>idRec</th>
<th>unionAttrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>{ Blackberry, Earthy, Silky, Tannins, Toast }</td>
</tr>
<tr>
<td>6</td>
<td>{ Berry, Plum, Robust, Zesty }</td>
</tr>
<tr>
<td>7</td>
<td>{ Cherry, Pepper, Prune, Silky, Tannins }</td>
</tr>
<tr>
<td>8</td>
<td>{ Cherry, Oak, Raspberry, Robust }</td>
</tr>
<tr>
<td>9</td>
<td>{ Earthy, Fruit, Strawberry, Silky, Tannins }</td>
</tr>
</tbody>
</table>

To take one set as an example, Record 5 has “Silky” and “Tannins” for its two Body assignments and “Blackberry”, “Earthy”, and “Toast” for its three Flavors assignments. The resulting set is a union of all five attribute values.

Set constructor

EQL allows users to write sets directly in queries.

The syntax of the set constructor is:

```
{ <expr1> [, <expr2>] * }
```

where the curly braces enclose a comma-separated list of one or more expressions.

For example, this is an integer set:

```
{ 1, 4, 7, 10 }
```

while this is a string set:

```
{ 'Red', 'White', 'Merlot', 'Chardonnay' }
```

Keep the following in mind when using set constructors:

- Set constructors may appear anywhere in a query where an expression is legal. (Because set constructors have a set type, you will get an EQL checking error if you use a set constructor in a context that expects an atomic value.)

- The individual elements of the set constructor may be arbitrary expressions, as long as they have the correct type. For instance, you may write the following as long as x, y, and z are integers:

```
{ x, y + z, 3 }
```

- All of the expressions within the curly braces must have the same type. For example, you cannot mix integers and strings.

- Empty set constructors are not allowed; there must be at least one expression within the curly braces.
Note that EQL promotes integers to doubles in a set constructor as needed. Therefore, writing \{1, 2\} results in an `mdex:long-set` type while \{1, 2.5\} results in an `mdex:double-set` type.

**Set constructor examples**

In this first example, the `SELECT` clause constructs a string-type set (named `selectWines`) that contains 'Red' and 'White' as its two elements. The `selectWines` set is then used in a `HAVING` clause to limit the returned records to those have `WineType` assignments of either 'Red' or 'White'.

```
RETURN results AS
SELECT
    {'Red', 'White'} AS selectWines,
    WineID AS idRec,
    WineType AS wines,
    Body AS bodyAttr
FROM WineState
HAVING wines IN selectWines
ORDER BY idRec
```

This second example is similar to the first example, except that the set is used in a `WHERE` clause:

```
RETURN results AS
SELECT
    WineID AS idRec,
    WineType AS wines,
    Body AS bodyAttr
FROM WineState
WHERE WineType IN {'Red', 'White'}
ORDER BY idRec
```

Both queries would return only records with a `WineType` of 'Red' or 'White'.

**Quantifiers**

EQL provides existential and universal quantifiers for use with Boolean expressions against sets. Both types of expressions can appear in any context that accepts a Boolean expression, such as `SELECT` clauses, `WHERE` clauses, `HAVING` clauses, `ORDER BY` clauses, join conditions, and so on.

**Existential quantifier**

An existential quantifier uses the `SOME` keyword. In an existential quantifier, if any item in the set has a match based on the comparison operator that is used, the returned value is `TRUE`.

The syntax of the existential quantifier is:

```
SOME id IN set SATISFIES (booleanExpr)
```

where:

- `id` is an arbitrary identifier for the item to be compared. The identifier must use the NCName format.
- `set` is a set of any set data type.
- `booleanExpr` is any expression that produces a Boolean (or NULL).

The expression binds the identifier `id` within `booleanExpr`. This binding shadows any other attributes with the same name inside the predicate. Note that this shadowing applies only to references to identifiers/attributes that do not have a statement qualifier.
To evaluate an existential quantifier expression, EQL evaluates the predicate expression for every member of
the indicated set. Then, EQL computes the results of the quantifier based on these predicate values as
follows:

1. If set is empty, the quantifier is FALSE.
2. Otherwise, if booleanExpr is true for least one element of set, the quantifier is TRUE.
3. Otherwise, if booleanExpr is false for every id element of set, the quantifier is FALSE.
4. Otherwise (the values of booleanExpr are either false or NULL, with at least one NULL), the quantifier is
   NULL.

Some results of this evaluation are:

- \( \text{SOME } x \text{ IN } \{ \} \text{ SATISFIES } (x > 0) \) is FALSE.
- \( \text{SOME } x \text{ IN } \{-3, -2, 1\} \text{ SATISFIES } (x > 0) \) is TRUE, because the predicate expression is true for \( x = 1 \).
- \( \text{SOME } x \text{ IN } \{5, 7, 10\} \text{ SATISFIES } (x > 0) \) is TRUE, because the predicate is true for \( x = 5 \).
- \( \text{SOME } x \text{ IN } \{ \text{’foo’, ’3’, ’4’} \} \text{ SATISFIES } (\text{TO_INTEGER}(x) > 0) \) is TRUE, because the predicate is true for \( x = ’3’ \).
- \( \text{SOME } x \text{ IN } \{ \text{’foo’, ’-1’, ’-2’} \} \text{ SATISFIES } (\text{TO_INTEGER}(x) > 0) \) is NULL. The predicate is false for \( x = ’-1’ \)
  and \( x = ’-2’ \), but NULL for \( x = ’foo’ \).

In this existential quantifier example, Body is a multi-assign string attribute (one of whose assignments on
several records is ‘Robust’):

```sql
RETURN results AS
SELECT
    WineID AS idRec,
    WineType AS wines,
    Body AS bodyAttr
FROM WineState
WHERE SOME x IN Body SATISFIES (x = ’Robust’)
ORDER BY idRec
```

The result of this statement would be:

<table>
<thead>
<tr>
<th>bodyAttr</th>
<th>idRec</th>
<th>wines</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Robust, Tannins }</td>
<td>4</td>
<td>Red</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>6</td>
<td>Red</td>
</tr>
<tr>
<td>{ Oak, Robust }</td>
<td>8</td>
<td>Red</td>
</tr>
<tr>
<td>{ Robust, Tannins }</td>
<td>11</td>
<td>Zinfandel</td>
</tr>
<tr>
<td>{ Fresh, Robust }</td>
<td>19</td>
<td>White</td>
</tr>
<tr>
<td>{ Firm, Robust }</td>
<td>22</td>
<td>Blanc de Noirs</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>23</td>
<td>Brut</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>24</td>
<td>Brut</td>
</tr>
<tr>
<td>{ Firm, Robust }</td>
<td>25</td>
<td>White</td>
</tr>
</tbody>
</table>

Only the nine records that have the Body='Robust' assignment are returned.

**Universal quantifier**

A universal quantifier uses the `EVERY` keyword. In a universal quantifier, if every item in the set has a match
based on the comparison operator that is used, the returned value is TRUE.

The syntax of the universal quantifier is:

```sql
EVERY id IN set SATISFIES (booleanExpr)
```
where `id`, `set`, and `booleanExpr` have the same meanings as in the existential quantifier.

The expression binds the identifier `id` within `booleanExpr`. This binding shadows any other attributes with the same name inside the predicate. Note that this shadowing applies only to references to identifiers/attributes that do not have a statement qualifier.

Similar to an existential quantifier expression, for a universal quantifier expression EQL evaluates the predicate expression for every member of the indicated set. Then, EQL computes the results of the quantifier based on these predicate values as follows:

1. If `set` is empty, the quantifier is TRUE.
2. Otherwise, if `booleanExpr` is false for at least one element of `set`, the quantifier is FALSE.
3. Otherwise, if `booleanExpr` is true for every element of `set`, the quantifier is TRUE.
4. Otherwise (the values of `booleanExpr` are either true or NULL, with at least one NULL), the quantifier is NULL.

Some results of this evaluation are:

- `EVERY x IN { } SATISFIES (x > 0)` is TRUE.
- `EVERY x IN { -3, -2, 1 } SATISFIES (x > 0)` is FALSE, because the predicate is false for `x = -3`.
- `EVERY x IN { 5, 7, 10 } SATISFIES (x > 0)` is TRUE, because the predicate is true for every value in the set.
- `EVERY x IN { 'foo', '3', '4' } SATISFIES (TO_INTEGER(x) > 0)` is NULL. The predicate is true for `x = '3'` and `x = '4'`, but NULL for `x = 'foo'`.
- `EVERY x IN { 'foo', '-1', '-2' } SATISFIES (TO_INTEGER(x) > 0)` is FALSE, because the predicate is false for `x = '-1'`.

This universal quantifier example is very similar to the existential quantifier example above:

```sql
RETURN results AS
SELECT
    WineID AS idRec,
    WineType AS wines,
    Body AS bodyAttr
FROM WineState
WHERE (EVERY x IN Body SATISFIES (x = 'Robust')) AND (WineID IS NOT NULL)
ORDER BY idRec
```

The result of this statement would be:

<table>
<thead>
<tr>
<th>bodyAttr</th>
<th>idRec</th>
<th>wines</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Robust }</td>
<td>1</td>
<td>Chardonnay</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>2</td>
<td>Chardonnay</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>6</td>
<td>Red</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>17</td>
<td>Merlot</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>20</td>
<td>White</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>21</td>
<td>White</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>23</td>
<td>Brut</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>24</td>
<td>Brut</td>
</tr>
</tbody>
</table>

The only records that are returned are those that have only one Body='Robust' assignment (Records 6, 23, and 24) and those that have no Body assignments (Records 1, 2, 17, 20, and 21).
In the query, note the use of the "WineID IS NOT NULL" expression in the \texttt{WHERE} clause. This prevents the return of other records in the system for which the universal expression would normally be evaluated as \texttt{TRUE} but which would return empty sets.

### Grouping by sets

EQL provides support for grouping by sets.

#### Using \texttt{GROUP BY}

In the normal grouping syntax for the \texttt{GROUP BY} clause, EQL groups by set equality (that is, rows for which the sets are equal are placed into the same group).

For example, assume a data set in which \texttt{Body} is a multi-assign attribute and every record has at least one \texttt{Body} assignment except for Records 1, 2, 17, 20, and 21. This query is made against that data set:

```sql
RETURN results AS
SELECT
  SET(WineID) AS IDs
FROM WineState
GROUP BY Body
```

The result of this statement might be:

<table>
<thead>
<tr>
<th>Body</th>
<th>IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ Fresh, Robust }</td>
<td>{ 1, 17, 20, 21 }</td>
</tr>
<tr>
<td>{ Supple }</td>
<td>{ 14, 15 }</td>
</tr>
<tr>
<td>{ Silky, Tannins }</td>
<td>{ 10, 12, 13, 16, 18, 3, 5, 7, 9 }</td>
</tr>
<tr>
<td>{ Firm, Robust }</td>
<td>{ 22, 25 }</td>
</tr>
<tr>
<td>{ Robust }</td>
<td>{ 23, 24, 6, 8 }</td>
</tr>
<tr>
<td>{ Robust, Tannins }</td>
<td>{ 11, 4 }</td>
</tr>
</tbody>
</table>

Keep in mind that when using \texttt{GROUP BY} that EQL preserves rows in which the group key is the empty set or a \texttt{NULL} value). Therefore, Records 1, 2, 17, 20, and 21 are returned even though they have no \texttt{Body} assignments (because the empty set is returned for those records).

For more information on the \texttt{GROUP BY} clause, see \textit{Specifying \texttt{GROUP BY} on page 33}.

#### Using \texttt{GROUP BY MEMBERS}

The \texttt{MEMBERS} extension to \texttt{GROUP BY} allows grouping by the members of a set. To illustrate the use of \texttt{MEMBERS}, the previous example can be re-written as:

```sql
RETURN results AS
SELECT
  SET(WineID) AS IDs
FROM WineState
GROUP BY MEMBERS(Body) AS BodyType
```

The result might be:

<table>
<thead>
<tr>
<th>BodyType</th>
<th>IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supple</td>
<td>{ 14, 15 }</td>
</tr>
<tr>
<td>Firm</td>
<td>{ 22, 25 }</td>
</tr>
<tr>
<td>Fresh</td>
<td>{ 19 }</td>
</tr>
<tr>
<td>Robust</td>
<td>{ 11, 19, 22, 23, 24, 25, 4, 6, 8 }</td>
</tr>
<tr>
<td>Tannins</td>
<td>{ 10, 11, 12, 13, 16, 18, 3, 4, 5, 7, 9 }</td>
</tr>
<tr>
<td>Silky</td>
<td>{ 10, 12, 13, 16, 18, 3, 5, 7, 9 }</td>
</tr>
<tr>
<td></td>
<td>{ 1, 17, 2, 20, 21 }</td>
</tr>
</tbody>
</table>

Note that like the previous example, Records 1, 2, 17, 20, and 21 are returned.

For more information on MEMBERS, see MEMBERS extension on page 34.
Chapter 6

EQL Use Cases

This section describes how to handle various business scenarios using EQL. The examples in this section are not based on a single data schema.

Re-normalization
Grouping by range buckets
Manipulating records in a dynamically computed range value
Grouping data into quartiles
Combining multiple sparse fields into one
Joining data from different types of records
Linear regressions in EQL
Using an IN filter for pie chart segmentation
Running sum
Query by age
Calculating percent change between most recent month and previous month

Re-normalization

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

In a sample data set, Employees source records were de-normalized onto Transactions, as shown in the following example:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DimEmployee_FullName</td>
<td>Tsvi Michael Reiter</td>
</tr>
<tr>
<td>DimEmployee_HireDate</td>
<td>2005-07-01T04:00:00.000Z</td>
</tr>
<tr>
<td>DimEmployee_Title</td>
<td>Sales Representative</td>
</tr>
<tr>
<td>FactSales_RecordSpec</td>
<td>SO49122-2</td>
</tr>
<tr>
<td>FactSales_SalesAmount</td>
<td>939.588</td>
</tr>
</tbody>
</table>
Incorrect

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

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

Correct

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

```eql
DEFINE Employees AS
SELECT
  ARB(DimEmployee_HireDate) AS DimEmployee_HireDate,
  ARB(DimEmployee_Title) AS DimEmployee_Title
FROM EmployeeState
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:

```eql
/**
 * 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 CT
FROM SaleState
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 CT
FROM SaleState
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:
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.
- Use an empty lookup list to get the range of interest into the desired expression.
- Use subtraction and HAVING to enable filtering by a dynamic value (HAVING must be used because Diff is not in scope in a WHERE clause on Result).

```
DEFINE ValueRange AS SELECT
    COUNT(1) AS CT
FROM SaleState
GROUP BY SalesAmount
HAVING SalesAmount > 1.0 AND SalesAmount < 10000.0;
RETURN Buckets AS SELECT
    SUM(CT) AS CT,
    FLOOR((SalesAmount - 1)/999.0) AS Bucket
FROM ValueRange
GROUP BY Bucket
ORDER BY Bucket
```

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,
```

Oracle® Big Data Discovery: EQL Reference

Version 1.0.0 • January 2015
```
SUM(FactSales_SalesAmount) AS Metric
FROM SaleState
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
```
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.

```
<table>
<thead>
<tr>
<th>ID</th>
<th>Make</th>
<th>Model</th>
<th>Type</th>
<th>leasePayment</th>
<th>loanPayment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Audi</td>
<td>A4</td>
<td>lease</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Audi</td>
<td>A4</td>
<td>loan</td>
<td></td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>BMW</td>
<td>325</td>
<td>lease</td>
<td>420</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BMW</td>
<td>325</td>
<td>loan</td>
<td></td>
<td>700</td>
</tr>
</tbody>
</table>
```

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

Joining data from different types of records

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

Use lookups against unfiltered records 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
```
The query is:

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

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

Linear regressions in EQL

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

Using the following data set:

<table>
<thead>
<tr>
<th>ID</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>60</td>
<td>3.1</td>
</tr>
<tr>
<td>2</td>
<td>61</td>
<td>3.6</td>
</tr>
<tr>
<td>3</td>
<td>62</td>
<td>3.8</td>
</tr>
<tr>
<td>4</td>
<td>63</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>65</td>
<td>4.1</td>
</tr>
</tbody>
</table>

The following simple formulation:

\[ y = A \pm Bx \]

Can be expressed in EQL as:

```sql
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
FROM DataState
GROUP
```
With the result:

<table>
<thead>
<tr>
<th>N</th>
<th>sumX</th>
<th>sumY</th>
<th>sumXY</th>
<th>sumX2</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>311.0000000</td>
<td>18.6000000</td>
<td>1159.7000000</td>
<td>19359.000000</td>
<td>0.187838</td>
<td>-7.963514</td>
</tr>
</tbody>
</table>

### 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
FROM DataState
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
FROM SaleState
GROUP BY Customer
ORDER BY Sales DESC
PAGE(0,5);
RETURN Others AS SELECT
  SUM(Sale) AS Sales
FROM SaleState
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 CalYear,
  DimDate_MonthNumberOfYear AS NumMonth,
  SUM(FactSales_SalesAmount) AS TotalSales
FROM SaleState
GROUP BY CalYear, NumMonth
ORDER BY CalYear DESC, NumMonth DESC
PAGE(0, 12);

RETURN CumulativeSum AS SELECT
  one.CalYear AS CalYear,
  one.NumMonth AS NumMonth,
  SUM(many.TotalSales) AS TotalSales
FROM Input one JOIN Input many
ON ((one.CalYear > many.CalYear) OR
  (one.CalYear = many.CalYear AND
   one.NumMonth >= many.NumMonth))
GROUP BY CalYear, NumMonth
ORDER BY CalYear, NumMonth

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
FROM SaleState
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
  ARB(DimDate_CalendarYear) AS CalYear,
  ARB(DimDate_MonthNumberOfYearOfYear) AS NumMonth,
  DimDate_CalendarYear * 12 + DimDate_MonthNumberOfYear AS OrdinalMonth,
  SUM(FactSales_SalesAmount) AS TotalSales
FROM SaleState
GROUP BY OrdinalMonth;
RETURN Result AS
SELECT
  CalYear AS CalYear,
  NumMonth AS NumMonth,
  TotalSales AS TotalSales,
  Input[OrdinalMonth - 1].TotalSales AS PriorMonthSales,
  100 * (TotalSales - PriorMonthSales) / PriorMonthSales AS PercentChange
FROM Input
ORDER BY CalYear DESC, NumMonth DESC
PAGE(0, 1)
Chapter 7
EQL Best Practices

This section discusses ways to maximize your EQL query performance.

Controlling input size

Filtering as early as possible

Controlling join size

Additional tips

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. When possible, use an already completed result from another statement instead of using collection 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:

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

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

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

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

---

Oracle® Big Data Discovery: EQL Reference

Version 1.0.0 • January 2015
FROM SalesState
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
FROM SaleState
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
FROM SaleState
WHERE FactSales_ProductKey IS NOT NULL
GROUP

In the example, FROM SalesState is the unique property key for the Sales collection.

### 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
FROM SaleState
GROUP BY g

The second form of the query filters the input (with the \( \text{WHERE} \) 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
FROM SaleState
WHERE (b < 10)
GROUP BY g

Another example of filtering records early is illustrated with the following pair of queries. Recall that a \( \text{WHERE} \) clause filters input records and a \( \text{HAVING} \) 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 \( g < 10 \).

```sql
RETURN Result AS SELECT SUM(a) AS sum_a FROM SaleState 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.

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

### Controlling join size

Joins can cause the Dgraph 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 Dgraph.

### 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.
- 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

A

about queries 10
ABS function 56
ADD_ELEMENT function 90
addition operator 56
aggregation
  function filters 46
  functions 59
  multi-level 45
  with APPROXCOUNTDISTINCT 44
  with COUNT 41
  with COUNT_APPROX 43
  with COUNTDISTINCT 43
APPROXCOUNTDISTINCT function 44
ARB function 74
arithmetic operators 70
AVG function 59

B

best practices
  additional tips 122
  controlling input size 120
  defining constants 121
  filtering as early as possible 121
BETWEEN operator 76
Boolean
  literal handling 50
  operators 70

C

calculate percent change over month 118
CARDINALITY function 91
CASE expression 76
case handling in EQL 51
CEIL function 56
characters in EQL 50
classes
  DEFINE 14
  FROM 20
  GROUP 32
  GROUP BY 32
  HAVING 26
  JOIN 21
  LET 15
  ORDER BY 27
  PAGE 30
  RETURN 15
  SELECT 17
  WHERE 25
  COALESCE expression 77
collections
    in FROM clause 20
    record sources 9
combining multiple sparse fields into one 115
commenting in EQL 11
CONCAT function 69
controlling input size 120
controlling join size 122
CORRELATION function 78
COS function 58
COUNT_APPROX function 43
COUNTDISTINCT function 43
COUNT function 41
CROSS JOIN 21
CUBE extension 39
cumulative sum 118
CURRENT_DATE function 63
CURRENT_TIMESTAMP function 63

D

data types 47
date and time values 61
  constructing 63
  using arithmetic operations on 68
DAY_OF_MONTH function 66
DAY_OF_WEEK function 66
DAY_OF_YEAR function 66
DEFINE clause 14
defining constants for best performance 121
DIFFERENCE function 92
DISTANCE function 61
division operator 56
double
  data type 48
  promotion from integer 53
EQL
case handling 51
characters 50
commenting 11
concepts 8
handling of inf results 54
handling of NaN results 54
handling of NULL results 52
inter-statement references 72
LOOKUP expressions 72
multi-level aggregation example 45
overview 8
processing order 11
reserved keywords 12
SQL comparison 9
syntax conventions 10
evaluation time and input size 120
EVERY function 107
existential quantifier 106
EXP function 57
expressions
CASE 76
COALESCE 77
GROUPING SETS 37
IN 79
in ORDER BY 28
LOOKUP 72
EXTRACT function 66
F
filtering 10
geocode 61
performance impact of 121
filters
per-aggregation 46
using results values as 72
FLOOR function 57
follow-on queries 72
FOREACH function 93
FROM clause 20
FROM_TZ function 65
FULL JOIN 21
functions
ABS 56
aggregation 59
APPROXCOUNTDISTINCT 44
ARB 74
arithmetic operators 70
AVG 59
CEIL 56
CONCAT 69
CORRELATION 78
COS 58
COUNT 41
COUNT_APPROX 43
COUNTDISTINCT 43
CURRENT_DATE 63
CURRENT_TIMESTAMP 63
date and time 61
DAY_OF_MONTH 66
day
DAY_OF_WEEK 66
day
DAY_OF_YEAR 66
distance 61
EXP 57
EXTRACT 66
FLOOR 57
FROM_TZ 65
GROUPING 41
HOUR 66
JULIAN_DAY_NUMBER 66
LATITUDE 61
LN 57
LOG 57
LONGITUDE 61
MAX 59
MEDIAN 59
MIN 59
MINUTE 66
MONTH 66
MONTH 66
numeric 56
PERCENTILE 80
POWER 58
QUARTER 66
RECORD_IN_FAST_SAMPLE 81
ROUND 57
SECOND 66
SIGN 58
SIN 58
SORT 58
STDDEV 60
string 69
STRING_JOIN 60
SUBSTR 70
TAN 58
TO_DATETIME 64
to
TO_DOUBLE 58
to
TO_DURATION 58, 64
to
TO_GEOCODE 61
to
TO_INTEGER 58
to
TO_STRING 70
to
TO_TIME 63
to
TO_TZ 65
to
TRUNC 58, 67
VARIANCE 60
WEEK 66
YEAR 66
G
gencode
data type 48
filtering 61
sorting by 28
GROUP BY clause 32
CUBE extension 39
MEMBERS extension 34
ROLLUP extension 38
GROUP clause 32
grouping
by range buckets 112
data into quartiles 113
GROUPING function 41
GROUPING SETS expression 37

multi-level aggregation example 45
multiplication operator 56

NaN, EQL handling of 54
NULL values
  and sets 84
  EQL handling of 52
numeric
  functions 56
  literal handling 50

operations, date and time 61
operators
  arithmetic 70
  Boolean 70
  precedence order 49
ORDER BY clause 27
order of processing in EQL 11
overview of queries 10

PAGE clause 30
  PERCENT modifier 30
  Top-K queries 30
PERCENTILE function 80
PERCENT modifier 30
pie chart segmentation with IN filters 117
POWER function 58
precedence rules for operators 49

queries 10
query by age 118
query processing order 11

RECORD_IN_FAST_SAMPLE function 81
re-normalization 111
reserved keywords 12
result values used as filters 72
RETURN clause 15
RIGHT JOIN 21
ROLLUP extension 38
ROUND function 57
running sum 118