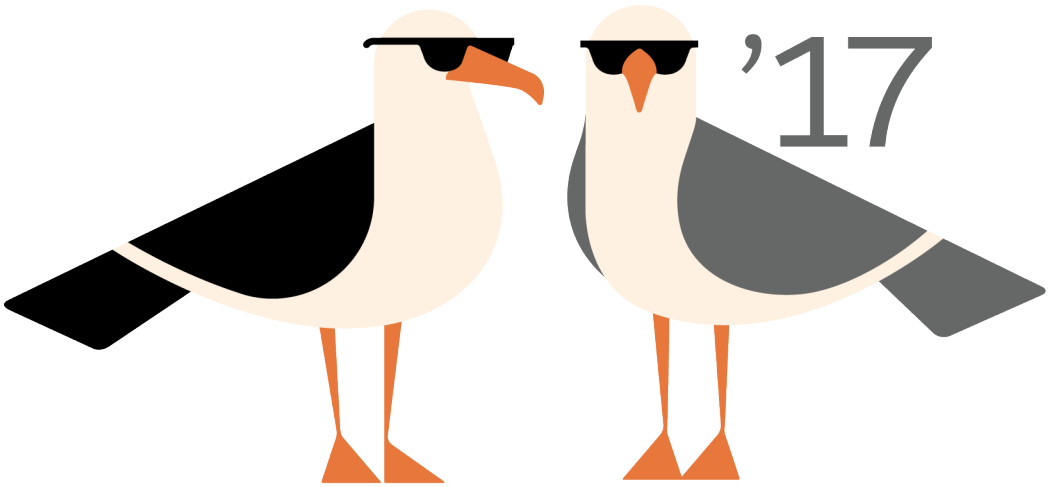
Analytics SAQL Reference

Salesforce, Summer '17



@salesforcedocs

Last updated: August 9, 2017

© Copyright 2000–2017 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

CONTENTS

SAQL OVERVIEW 1

ENABLE SAQL LOGS IN THE BROWSER 2

SAQL BASIC ELEMENTS 3
Statements 3
Keywords 3
Identifiers 4
Number Literals 4
String Literals 4
Boolean Literals 5
Quoted String Escape Sequences 5
Special Characters 5
Comments 6
Nulls and Nulls in Measures 6

SAQL OPERATORS 8
Arithmetic Operators 8
Comparison Operators 8
String Operators 10
Logical Operators 10
case 11
Null Operators 14

SAQL STATEMENTS 16
load 16
filter 16
foreach 17
group and cogroup 18
union 21
order 21
limit 22
offset 23

SAQL FUNCTIONS 24
Aggregate Functions 24
Date Functions 28
String Functions 37
Math Functions 40

Windowing Functions 42
coalesce() 49

QUERY PERFORMANCE 51
Projection is Key 53
Grouping Order 54
Network Traffic and Latency 54
Redundant Filters 54
Use the ELT Process 55
Multi-Value Dimensions 56

SAQL OVERVIEW

Use SAQL (Salesforce Analytics Query Language) to access data in Analytics datasets. Analytics uses SAQL behind the scenes in lenses, dashboards, and explorer to gather data for visualizations.

Developers can write SAQL to directly access Analytics data via:

- Analytics REST API

  Build your own app to access and analyze Analytics data or integrate data with existing apps.

- Dashboard JSON

  Create advanced dashboards. A dashboard is a curated set of charts, metrics, and tables.

SEE ALSO:

Wave REST API Developer's Guide

Wave Analytics Dashboard JSON Reference

ENABLE SAQL LOGS IN THE BROWSER

If you're using Google Chrome to work with SAQL and Salesforce Analytics, you can turn on SAQL logs.

Turning on SAQL logs in the browser prints queries in the Developer Tools Console. It doesn't change server-side logs.

1. In Google Chrome, open Developer Tools.
2. Select Console.
3. Select the explore (wave.apexp) frame.
4. Enter `edge.log.enabled = true`.
5. Enter `edge.log.query = true`.

SAQL BASIC ELEMENTS

Statements

A SAQL query loads an input dataset, operates on it, and outputs a results dataset. A query is made up of statements. Each SAQL statement has an input stream, an operation, and an output stream.

A statement is made up of keywords (such as `filter`, `group`, and `order`), identifiers, literals, and special characters. Statements can span multiple lines and must end with a semicolon.

Assign each query line to an identifier called a *stream*. The only exception to this rule is the last line in a query, which you don't need to assign explicitly.

The output stream is on the left side of the `=` operator and the input stream is on the right side of the `=` operator.

Example: Each of the lines in this SAQL query is a SAQL statement:

```
q = load "0Fcc00000004DI1CAM/0Fd500000004F4sCAE";
q = group q by all;
q = foreach q generate count() as 'count', unique('OL.Helpful') as 'unique_OL.Helpful';
limit q 2000;
```

SAQL is compositional—you can chain statements together to operate on data sequentially. The order of SAQL statements is enforced according to how the operations in the statements change the results of a query.

The statement order rules:

- The order of `filter` and `order` can be swapped because it doesn't change the results.
- `offset` must be after `filter` and `order`
- `offset` must be before `limit`
- There can be no more than 1 `offset` statement after a `foreach` statement.

Tip: SAQL is influenced by the Pig Latin programming language, but their implementations differ and they aren't compatible.

SEE ALSO:

filter
foreach
limit
offset
order

Keywords

Keywords are case-sensitive and must be lowercase.

Identifiers

SAQL identifiers are case-sensitive. They can be enclosed in single quotation marks (') or no quotation marks.

Quoted identifiers can contain any character that a string can contain.

Unquoted identifiers can't be a reserved words and must start with a letter (A to Z or a to z) or an underscore. Subsequent characters can be letters, numbers, or underscores. Unquoted identifiers can't contain spaces.

This example uses valid syntax:

```
accounts = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
opps = load "0Fcyy000000002qCAA/0Fcyy000000002WCAQ";
c = group accounts by 'Year', opps by 'Year';
d = foreach c generate opps.Year as 'Year';
e = filter d by Year == "2002";
```

In the following example, the code in the third line throws an error:

```
accounts = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
opps = load "0Fcyy000000002qCAA/0Fcyy000000002WCAQ";
c = group accounts by "Year", opps by "Year";
d = foreach c generate opps.Year as 'Year';
e = filter d by Year == "2002";
```

Note: A set of characters in double quotes is treated as a string rather than as an identifier.

Number Literals

A number literal represents a number in your script.

Some examples of number literals are 16 and 3.14159. You can't explicitly assign a type (for example, integer or floating point) to a number literal. Scientific E notation isn't supported.

The responses to queries are in JSON. Therefore, the returned numeric field is a "number" class.

String Literals

A string is a set of characters inside double quotes (").

Example: `"This is a string."`

This example uses valid syntax:

```
accounts = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
opps = load "0Fcyy000000002qCAA/0Fcyy000000002WCAQ";
c = group accounts by 'Year', opps by 'Year';
d = foreach c generate opps.Year as 'Year';
e = filter d by Year == "2002";
```

Note: Identifiers are either unquoted or enclosed in single quotation marks.

Boolean Literals

A boolean literal represents true or false (yes or no) in your script.

Boolean literals `true` and `false` are supported in SAQL.

Quoted String Escape Sequences

Strings can be escaped with the backslash character.

You can use the following string escape sequences:

Sequence	Meaning
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\'`	One single-quote character
`\"`	One double-quote character
`\\`	One backslash character

Special Characters

Certain characters have special meanings in SAQL.

Character	Name	Description
`;`	Semicolon	Used to terminate statements.
`'`	Single quote	Used to quote identifiers.
`"`	Double quote	Used to quote strings.
`()`	Parentheses	Used for function calls, to enforce precedence, for order clauses, and to group expressions. Parentheses are mandatory when you're defining more than one group or order field.
`[]`	Brackets	Used to denote arrays. For example, this is an array of strings: `[ "this", "is", "a", "string", "array" ]` Also used for referencing a particular member of an object. For example, `em['miles']`, which is the same as `em.miles`.
`.`	Period	Used for referencing a particular member of an object. For example, `em.miles`, which is the same as `em['miles']`.

Character	Name	Description
`::`	Two colons	Used to explicitly specify the dataset that a measure or dimension belongs to, by placing it between a dataset name and a column name. Using two colons is the same as using a period (.) between names. For example: `data = foreach data generate left::airline as airline`
`..`	Two periods	Used to separate a range of values. For example: `c = filter b by "the_date" in ["2011-01-01".."2011-01-31"];`

Comments

Two sequential hyphens (--) indicate the beginning of a single-line comment in SAQL.

You can put a comment on its own line:

```
--Load a data stream.
a = load "myData";
```

You can put a comment at the end of a line:

```
a = load "myData"; --Load a data stream.
```

You can comment out a SAQL statement:

```
--The following line is commented out:
--a = load "myData";
```

Nulls and Nulls in Measures

In most contexts, SAQL allows the use of null anywhere a constant string or number would appear. SAQL also supports use of null measures.

Using Nulls In SAQL

You can specify a `null` constant almost anywhere a constant string or number can appear in SAQL, with the following exceptions and clarifications.

Typing

`null` is not typed. It is inferred from context. For example, `null` + 4 is a number. A SAQL syntax error will be generated when a type cannot be infered.

Filters

When a filter expression evaluates to `null`, the row is filtered out.

- **Lists**

  `foo in [null, "bar"]` is handled like `foo == null` or `foo == "bar"`.

- **Ranges**

`filter q by dim in [null.."myvalue"]` is handled as `(dim>=null and dim<=7)`

Unsupported

`null` is not supported in the following contexts:

- Offset
- Limit
- dateRelative
- dateRange
- Windowing range
- Trim (second argument)

Null Values in Measures

Measures in Analytics are dataset columns that contain numerical values. Analytics supports null values in measures.

Note: If null measure handling is not enabled in your org, it can be enabled by your admin.

Null measure handling lets customers distinguish between null and non-null—for example, the number 0—values in their numerical data. SAQL support for null measures facilitates this customer preference; for example, when using aggregation, comparison, and math functions, and for `order by` or `group by` clauses.

When you create or update a dataset, for example through your dataflow or a CSV upload, any blank measure values in your data are replaced with specific values. Analytics uses the default values specified in your dataflow or CSV metadata file to replace blank values.

Replacing blank values with zeros can be problematic for a number of reasons. Take the example of data with customer satisfaction scores, where some customers have not responded. Calculated values such as average and minimum are correct in the source data, but when blank values are replaced with zeros when the dataset is created, the resulting calculations are incorrect.

Null measure handling lets you specify defaults using the special "null" value in your dataflows and CSV metadata files. When no default value is specified, Analytics replaces blanks with null values.

For more information on null measure handling and how to set it up, see the Analytics Data Integration Guide.

SAQL OPERATORS

Use operators to perform mathematical calculations or comparisons.

Arithmetic Operators
Use arithmetic operators to perform addition, subtraction, multiplication, division, and modulo operations.

Comparison Operators
Use comparison operators to compare values.

String Operators
To concatenate strings, use the plus sign (+).

Logical Operators
Use logical operators to perform AND, OR, and NOT operations.

case
Use the SAQL `case` operator within a `foreach` statement to create logic that chooses between conditions. The `case` operator supports two syntax forms: searched case expression and simple case expression.

Null Operators
Use null operators to test whether a value is null.

Arithmetic Operators

Use arithmetic operators to perform addition, subtraction, multiplication, division, and modulo operations.

Operator	Description
+	Plus
–	Minus
*	Multiplication
/	Division
%	Modulo

Comparison Operators

Use comparison operators to compare values.

Comparisons are defined for values of the same type only. For example, strings can be compared with strings and numbers compared with numbers.

Operator	Name	Description
`==`	Equals	`True` if the operands are equal. String comparisons that use the equals operator are case-sensitive.
`!=`	Not equals	`True` if the operands aren't equal.
`<`	Less than	`True` if the left operand is less than the right operand.
`<=`	Less or equal	`True` if the left operand is less than or equal to the right operand.
`>`	Greater than	`True` if the left operand is greater than the right operand.
`>=`	Greater or equal	`True` if the left operand is greater than or equal to the right operand.
`matches`	Matches	`True` if the left operand contains the string on the right. Wildcards and regular expressions aren't supported. This operator is not case-sensitive. For example, the following query matches airport codes such as LAX, LAS, ALA, and BLA: `my_matches = filter a by origin matches "LA";`
`in`	In	If the left operand is a dimension, `true` if the left operand has one or more of the values in the array on the right. For example: `a1 = filter a by origin in ["ORD", "LAX", "LGA"];` If the left operand is a measure, `true` if the left operand is in the array on the right. You can use the `date()` function to filter by date ranges. If the array is empty, everything is filtered and the results are empty. Ranges that are out of order (for example, `in ["20 years ago" .. "2016-01-11"]` or `in ["Z" .. "A"]`), evaluate to `false`.
`not in`	Not in	`True` if the left operand isn't equal to any of the values in an array on the right. The results include rows for which the origin key doesn't exist. For example: `a1 = filter a by origin not in ["ORD", "LAX", "LGA"];`

Example: Given a row for a flight with the origin "SFO" and the destination "LAX" and weather of "rain" and "snow," here are the results for each type of "in" operator:

```
weather in ["rain", "wind"] = true
weather not in ["rain", "wind"] = false
```

SEE ALSO:

filter

String Operators

To concatenate strings, use the plus sign (+).

Operator	Description
+	Concatenate

Example: To combine the year, month, and day into a value that's called `CreatedDate`:

```
q = foreach q generate Id as Id, Year + "-" + Month + "-" + Day as CreatedDate;
```

Logical Operators

Use logical operators to perform AND, OR, and NOT operations.

Logical operators can return true, false, or null.

Operator	Name	Description
`&& (and)`	Logical AND	See table.
`|| (or)`	Logical OR	See table.
`! (not)`	Logical NOT	See table.

The following tables show how nulls are handled in logical operations.

x	y	x && y	x || y
True	True	True	True
True	False	False	True
True	Null	Null	True
False	True	False	True
False	False	False	False
False	Null	False	Null
Null	True	Null	True
Null	False	False	Null
Null	Null	Null	Null

x	!x
True	False

x	!x
False	True
Null	Null

case

Use the SAQL `case` operator within a `foreach` statement to create logic that chooses between conditions. The `case` operator supports two syntax forms: searched case expression and simple case expression.

Syntax—Searched Case Expression

```
case
      when search_condition then result_expr
      [ when search_condition2 then result_expr2 … ]
      [ else default_expr ]
end
```

`case...end`

The `case` and `end` keywords begin and close the expression.

`when...then`

The `when` and `then` keywords define a conditional statement. A `case` expression can contain one or more conditional statement.

- *`search_condition`*—Any logical expression that can be evaluated to `true` or `false`. This expression may be constructed using any values, identifiers, logical operator, comparison operator, or scalar functions (including date and math functions) supported by SAQL. Examples of valid *`search_condition`* syntax:
  - `xInt < 5`
  - `price > 1000 and price <= 2000`
  - `units*round(price_per_unit) < abs(revenue)`
- *`result_expr`*—Any expression that can be evaluated by the SAQL engine. May contain values, identifiers, and scalar functions (including date and math functions). The expression may evaluate to any data type. However, this data type must be consistent among all conditional expressions. That is, if *`result_expr`* is of NUMERIC type, then *`result_expr2`* … *`result_exprN`* must be of NUMERIC type. Examples of valid *`result_expr`* syntax:
  - `xInt`
  - `toString('orderDate', "dd/MM/yyyy")`
  - `"abc"`

`else`

(Optional)—Allows a default expression to be specified. The `else` statement must follow the conditional `when/then` statement. There can be only one `else` statement.

- *`default_expr`*—Any expression that can be evaluated by the SAQL engine. May contain values, identifiers, and scalar functions (including date and math functions). The data type must be consistent with the data type of *`result_expr`* specified in the preceding conditional statements.

Usage—Searched Case Expression

Conditional statements are evaluated on a row by row basis in the order in which they are given. If a *search_condition* evaluates as `true`, the corresponding *result_expr* is returned for that row. Therefore, if more than one of the conditional statements returns `true`, only the first one is evaluated. At least one `when/then` statement must be provided. An unlimited number of `when/then` statements may be provided.

A *default_expr* may be set with the optional `else` statement. If none of the *search_condition* expressions evaluate to `true`, the *default_expr* expression is returned. If no `else` statement is specified, `null` is returned as the default.

Syntax—Simple Case Expression

```
case primary_expr
      when test_expr then result_expr
      [ when test_expr2 then result_expr2 … ]
      [ else default_expr ]
end
```

`case...end`

The `case` and `end` keywords begin and close the expression.

- *primary_expr*—Any scalar expression that can be evaluated by the SAQL engine. May contain values, identifiers, and scalar functions (including date and math functions). The expression may evaluate to any comparable data type (NUMERIC, STRING, or DATE). Examples of valid *primary_expr* syntax:
  - `xInt % 3`
  - `date('year', 'month', 'day')`
  - `"abc"`

Note: A *scalar expression* takes single values as input and outputs single values. When used with `case`, the input values can be any expression that is valid in the context of a `foreach` statement.

`when...then`

The `when` and `then` keywords define a conditional statement. A `case` expression can contain one or more conditional statements.

- *test_expr*—Any scalar expression that can be evaluated by the SAQL engine. This expression may be constructed using any values, identifiers, and scalar functions (including date and math functions), but must evaluate to the same data type as the *primary_expr*. Examples of valid *test_expr* syntax:
  - `5`
  - `"abc"`
  - `abs(profit)`
- *result_expr*—Any scalar expression that can be evaluated by the SAQL engine. May contain values, identifiers, and scalar functions (including date and math functions). The expression may evaluate to any data type. However, this data type must be consistent among all conditional statements. That is, if *result_expr* is of NUMERIC type, then *result_expr2...result_exprN* must be of NUMERIC type. Examples of *result_expr* syntax:
  - `xInt`
  - `toString('orderDate', "dd/MM/yyyy")`
  - `"abc"`

`else`

(Optional) The `else` keyword allows a default expression to be specified. The `else` statement must follow conditional `when/then` statements. There can be only one `else` statement.

- *`default_expr`*—Any scalar expression that can be evaluated by the SAQL engine. May contain values, identifiers, and scalar functions (including date and math functions). The data type must be consistent with the data type of *`result_expr`* specified in the preceding conditional statements.

Usage—Simple Case Expression

Conditional statements are evaluated on a row by row basis in the order that they are given. If ***`primary_expr == test_expr`*** for a given conditional statement, the corresponding *`result_expr`* is returned for that row. At least one `when/then` statement must be provided. An unlimited number of `when/then` statements may be provided.

A *`default_expr`* may be set with the optional `else` statement. If *`primary_expr`* doesn't equal any of the *`test_expr`* conditions, the *`default_expr`* is returned. If no `else` statement is specified, `null` is returned as the default.

Tip: This simple case expression syntax is shorthand for a common instance of the searched case expression syntax. The first block of code is simple case expression syntax and the second block of code is searched case expression syntax. Both blocks of code have the same meaning.

```
case primary_expr
      when test_expr then result_expr
      when test_expr2 then result_expr2
else default_expr
```

```
case
      when primary_expr == test_expr then result_expr
      when primary_expr2 == test_expr2 then result_expr2
else default_expr
```

Using `case` Statements

Use `case` expressions in `foreach` clauses. Don't use `case` expressions in `order by`, `group by`, or `filter by` clauses.

Example: This example query uses the simple case expression syntax:

```
q = load "data";
q = foreach q generate xInt, (case xInt % 3
      when 0 then "3n"
      when 1 then "3n+1"
      else "3n+2"
end) as modThree;
```

Example: This example query uses the searched case expression syntax:

```
q = load "data";
q = foreach q generate price, (case
      when price < 1000 then "category1"
      when price >= 1000 and price < 2000 then "category2"
      else "category3"
end) as priceLevel;
```

Handling Null Values

In general, `null` values can't be compared. When *`search_condition`*, *`primary_expr`*, or *`test_expr`* evaluates to `null`, the *`default_expr`* specified by `else` (or `null` if no `else` clause is provided) is returned. For instance, the following query returns `"Other"` whenever `Mea1` evaluates to `null`:

```
q = load "data";
q = foreach q generate Mea1, (case Mea1
      when 0 then "Type1"
      when 1 then "Type2"
      else "Other"
end) as Category;
```

However, it is possible to specifically a condition on a `null` value by using the `is null` and `is not null` operations.

```
q = load “data”;
q = foreach q generate Mea1, (case
      when Mea1 is null then "Is Null"
      else "Is Not Null"
end) as Category;
```

Best Practices for Working with Dates

Before you use date values in `case` expressions, use the SAQL `toDate()` function to convert the date values from strings or Unix epoch seconds. Doing so ensures the most consistent comparisons.

Example:

```
q = load "data/dates";
q = foreach q generate OrderDate, (case
      when toDate(OrderDate_epoch_secs) < toDate("2/1/2015", "M/d/yyyy") and
toDate(OrderDate_epoch_secs) >= toDate("1/1/2015", "M/d/yyyy") then "Jan"
      else "Other"
end) as Month;
```

SEE ALSO:

foreach

Null Operators

Use null operators to test whether a value is null.

Null operators can return true or false.

Operator	Name	Description
`is null`	is null	True when the value is null.
`is not null`	is not null	True when the value is not null.

Note: `is null` and `is not null` can be used in projections, and in post-projection filters.

These are valid examples:

```
a = load "dataset";
b = foreach a generate Name as Name, Year as Year;
c = filter b by Year is not null;
```

```
q = load "dataset";
q = foreach q generate (case when Name is null then "john doe" else Name end) as Name;
```

This is **not** a valid example:

```
a = load "dataset";
a = filter a by Year is not null;
a = foreach a generate Name as Name, Year as Year;
```

SAQL STATEMENTS

load

Loads a dataset. All SAQL queries start with a `load` statement.

Syntax

```
result = load dataset;
```

If you're working in Dashboard JSON, *`dataset`* must be the dataset name from the UI. Use of the dataset name (also called an *alias*) means the app can substitute it with the correct version of the dataset.

If you're working in the Analytics REST API, *`dataset`* must be the containerId/versionId.

Usage

After being loaded, the data is in ungrouped form. The columns are the columns of the loaded dataset.

Example: The following example loads the dataset with ContainerID "0Fbxx000000002qCAA" and VersionID "0Fcxx000000002WCAQ" to a stream named "b": `b = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";`

Example: The following example loads the dataset with the name "Accounts" to a stream named "b": `b = load "Accounts";`

filter

Selects rows from a dataset based on a filter condition called a *predicate*.

Syntax

```
result = filter rows by predicate;
```

Usage

A predicate is a Boolean expression that uses comparison or logical operators. The predicate is evaluated for every row. If the predicate is `true`, the row is included in the result. Comparisons on dimensions are lexicographic, and comparisons on measures are numerical.

When a filter is applied to grouped data, the filter is applied to the rows in the group. If all member rows are filtered out, groups are eliminated. You can run a `filter` statement before or after `group` to filter out members of the groups.

Note: With results binding, an error may occur if the results from a previous step exceed the values supported by SAQL. For example, if something like `filter q by dim1 in {{results(Step_1)}};` produces a filter tree with a depth greater than 10,000 values, SAQL will fail with an error.

Example: The following example returns only rows where the origin is ORD, LAX, or LGA: `a1 = filter a by origin in ["ORD", "LAX", "LGA"];`

Example: The following example returns only rows where the destination is LAX or the number of miles is greater than 1,500: `y = filter x by dest == "LAX" || miles > 1500;`

Example: When `in` operates on an empty array in a `filter` operation, everything is filtered and the results are empty. The second statement filters everything and returns empty results:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
a = filter a by Year in [];
c = group a by ('Year', 'Name');
d = foreach c generate 'Name' as 'group::AName', 'Year' as 'group::Year',
sum(accounts::Revenue) as 'sRev';
```

SEE ALSO:

- Comparison Operators
- Logical Operators
- Statements

foreach

Applies a set of expressions to every row in a dataset. This action is often referred to as *projection*.

Syntax

```
q = foreach q generate expression as alias[, expression as alias ...];
```

The output column names are specified with the `as` keyword. The output data is ungrouped.

Using foreach with Ungrouped Data

When used with ungrouped data, the `foreach` statement maps the input rows to output rows. The number of rows remains the same.

Example: `a2 = foreach a1 generate carrier as carrier, miles as miles;`

Using foreach with Grouped Data

When used with grouped data, the `foreach` statement behaves differently than it does with ungrouped data.

Fields can be directly accessed only when the value is the same for all group members. For example, the fields that were used as the grouping keys have the same value for all group members. Otherwise, use aggregate functions to access the members of a group. The type of the column determines which aggregate functions you can use. For example, if the column type is numeric, you can use the `sum()` function.

Example: `z = foreach y generate day as day, unique(origin) as uorg, count() as n;`

Using `foreach` with a `case` Expression

To create logic in a `foreach` statement that chooses between conditional statements, use a `case` expression.

Example: This example query uses the simple case expression syntax:

```
q = load "data";
q = foreach q generate xInt, (case xInt % 3
      when 0 then "3n"
      when 1 then "3n+1"
      else "3n+2"
end) as modThree;
```

Example: This example query uses the searched case expression syntax:

```
q = load "data";
q = foreach q generate price, (case
      when price < 1000 then "category1"
      when price >= 1000 and price < 2000 then "category2"
      else "category3"
end) as priceLevel;
```

Use Unique Names

Using a name multiple times in a projection throws an error.

For example, the last line in this query is invalid and throws an error:

```
l = load "0Fabb000000002qCAA/0Fabb000000002WCAQ";
r = load "0Fcyy000000002qCAA/0Fcyy000000002WCAQ";
l = foreach l generate 'value'/'divisor' as 'value' , category as category;
r = foreach r generate 'value'/'divisor' as 'value' , category as category;
cg = cogroup l by category right, r by category;
cg = foreach cg generate r.category as 'category', sum(r.value) as sumrval, sum(l.value)
as sumrval;
```

SEE ALSO:

- Statements
- Aggregate Functions
- case

`group` and `cogroup`

Groups matched records. The `group` and `cogroup` statements are interchangeable. However, `cogroup` is typically used to operate on more than 1 input stream.

Syntax

```
result = group rows by field;
result = group rows by (field1, field2, ...);
result = group rows by expression[, rows by expression ...];
result = group rows by expression [left | right | full], rows by expression;
```

Simple Grouping

Adds one or more columns to a group. If data is grouped by a value that's `null` in a row, that whole row is removed from the result.

Syntax:

```
result = group rows by field;
```

or

```
result = group rows by (field1, field2, ...);
```

Note: The order of the fields matters for limit queries, but not for top queries.

Group by 1 dimension:

```
a = group a by year;
```

Group by multiple dimensions:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
a = group a by (year, month);
a = foreach a generate year as year, month as month;
```

Inner Cogrouping

Cogrouping means that two input streams, called *left* and *right* are grouped independently and arranged side by side. Only data that exists in both groups appears in the results.

Syntax:

```
result = cogroup rows by expression[, rows by expression ...];
```

This example is a simple `cogroup` operation on 2 datasets:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fbyy000000002qCAA/0Fcyy000000002WCAQ";
a = cogroup a by carrier, b by carrier;
```

You can cogroup more than 2 datasets:

```
result = cogroup a by keya, b by keyb, c by keyc;
```

This example performs a `cogroup` operation:

```
z = cogroup x by (day,origin), y by (day,airport);
```

You can't have the same stream on both sides of a `cogroup` operation. To perform a `cogroup` operation on 1 dataset, load the dataset twice so you have 2 streams.

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = cogroup a by ClosedDate, b by CreatedDate;
c = foreach b generate sum(a.Amount) as Amount;
```

You can also load 1 dataset and filter it into 2 different streams:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
a = filter a by "region" in ["West"];
a = filter a by "status" in ["closed"];
b = filter a by "year" in [2014];
c = filter a by "year" in [2015];
d = cogroup b by ("state"), c by ("state");
d = foreach d generate "state" as "state", sum(b.Amount) as "Amount_2014", sum(c.Amount)
as "Amount_2015";
```

This code throws an error because it performs a `cogroup` operation on a single stream, `a`:

```
 a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = cogroup a by ClosedDate, a by CreatedDate;
c = foreach b generate sum(a.Amount) as Amount;
```

To use aggregate functions when cogrouping, specify which input side to use in the aggregate function. For example, if you have an `a` side and a `b` side, and each contains a particular measure, use one of these syntaxes:

```
sum(inputSide['myMeasure'])
sum(inputSide::myMeasure)
sum(inputSide.myMeasure)
```

This query is valid because it uses the third syntax form to specify that `miles` comes from the `a` side.

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fbyy000000002qCAA/0Fcyy000000002WCAQ";
c = cogroup a by x, b by y;
d = foreach c generate a.x as x, a.y as y, sum(a.miles) as miles;
```

This query isn't valid because `miles` doesn't specify which side it is coming from:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fbyy000000002qCAA/0Fcyy000000002WCAQ";
c = cogroup a by x, b by y;
d = foreach c generate a.x as x, a.y as y, sum(miles) as miles;
```

If a lens or dashboard has a `cogroup` query, specify the input stream for projections and for `count()` aggregations on `cogroup` queries, as in this example:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fbyy000000002qCAA/0Fyy000000002WCAQ";
c = cogroup a by 'OwnerName', b by 'OwnerName';
c = foreach c generate a['OwnerName'] as 'OwnerName', sum(a['AmountConverted']) /
    sum(b['Amount']) as 'sum_target_completed', count(a) as count;
```

Outer Cogrouping

Outer cogrouping combines groups as an outer join. For the half-matches, null rows are added. The grouping keys are taken from the input that provides the value.

Syntax:

```
result = cogroup rows by expression [left | right | full], rows by expression;
```

Specify `left`, `right`, or `full` to indicate whether to perform a left outer join, a right outer join, or a full join.

Example: `z = cogroup x by (day,origin) left, y by (day,airport);`

You can apply an outer cogrouping across more than 2 sets of data. This example does a left outer join from a to b, with a right join to c:

```
result = cogroup a by keya left, b by keyb right, c by keyc;
```

Note: Outer joins return null when there is no match, instead of defaulting to zero.

union

Combines multiple result sets into one result set.

Syntax

```
result = union resultSetA, resultSetB [, resultSetC ...];
```

order

Sorts in ascending or descending order on one or more fields.

Syntax

```
result = order rows by field [ asc | desc ];
result = order rows by (field [ asc | desc ], field [ asc | desc ]);
```

`asc` or `desc` specifies whether the results are ordered in ascending (`asc`) or descending (`desc`) order. The default order is ascending.

Usage

The `order` statement isn't applied to the whole set. The `order` statement operates on rows individually.

You can use the `order` statement with ungrouped data. You can also use the `order` statement to specify order within a group or to sort grouped data by an aggregated value.

Example: `q = order q by 'count' desc;`

Example: To `order` a stream by multiple fields, use this syntax:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = group a by (year, month);
c = foreach b generate year as year, month as month;
d = order c by (year desc, month desc);
```

Example: You can order a cogrouped stream before a `foreach` statement:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = load "0Fayy000000002qCAA/0Fbyy000000002WCAQ";
c = cogroup a by year, b by year;
c = order c by a.airlineName;
c = foreach c generate year as year;
```

Example: You can't reference a preprojection ID in a postprojection `order` operation. (*Projection* is another term for a `foreach` operation.) This code throws an error:

```
q = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
q = group q by 'FirstName';
q = foreach q generate sum('mea_mm10M') as 'sum_mm10M';
q = order q by 'FirstName' desc;
```

This code is valid:

```
q = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
q = group q by 'FirstName';
q = foreach q generate 'FirstName' as 'User_FirstName', sum('mea_mm10M') as 'sum_mm10M';
q = order q by 'User_FirstName' desc;
```

SEE ALSO:

Statements

limit

Limits the number of results that are returned. If you don't set a limit, queries return a maximum of 10,000 rows.

Syntax

```
result = limit rows number;
```

Usage

Use this statement only on data that has been ordered with the `order` statement. The results of a `limit` operation aren't automatically ordered, and their order can change each time that statement is called.

You can use the `limit` statement with ungrouped data.

You can use the `limit` statement to limit grouped data by an aggregated value. For example, to find the top 10 regions by revenue: group by region, call `sum(revenue)` to aggregate the data, `order` by `sum(revenue)` in descending order, and `limit` the number of results to the first 10.

Note: The `limit` statement isn't a `top()` or `sample()` function.

Example: This example limits the number of returned results to 10:

```
b = limit a 10;
```

The expression can't contain any columns from the input. For example, this query is not valid:

```
b = limit OrderDate 10;
```

SEE ALSO:

Statements

order

offset

Paginates values from query results.

Syntax

```
result = offset rows number;
```

Usage

Used to paginate values from query results. This statement requires that the data has been ordered with the `order` statement.

Example: This example loads a dataset, puts the rows in descending order, and returns rows 400 to 800:

```
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = foreach a generate 'carrier' as 'carrier', count() as 'count';
c = order b by 'count' desc;
d = limit c 400;
e = offset d 400;
```

SEE ALSO:

Statements

SAQL FUNCTIONS

Aggregate Functions

Use aggregate functions to perform computations on values.

Using an aggregate function on an empty set returns null. For example, if you use an aggregate function with a nonmatching column of an outer cogrouping, you might have an empty set.

Aggregation functions treat each line as its own group if not preceded by `group by`.

This table lists the aggregate functions that are supported:

Aggregate Function	Description

`avg()` or `average()`	Returns the average value of a numeric field.
	For example, to calculate the average number of miles:
	`a1 = group a by (origin, dest);`
	`a2 = foreach a1 generate origin as origin, dest as destination, average(miles) as miles;`

`count()`	Returns the number of rows that match the query criteria.
	For example, to calculate the number of carriers:
	`q = foreach q generate 'carrier' as 'carrier', count() as 'count';`
	The `count()` function operates on streams that were inputs to the `group` or `cogroup` statements. It doesn't operate on the newly grouped stream or on an ungrouped stream.
	`a = load "0Fcyy000000002qCAA/0Fcyy000000002WCAQ";`
	`a1 = group a by (Year);`
	`q = foreach a1 generate` **`count(a)`** `as countYear, count() as count, Year as year;`
	`q = limit q 20;`
	You can't pass `a1` to the `count()` function because it's a newly grouped stream.

`first()`	Returns the value for the first tuple. To work as expected, you must be aware of the sort order or know that the values of that measure are the same for all tuples in the set.
	For example, you can use these statements to compute the distance between each combination of origin and destination:
	`a1 = group a by (origin, dest);`
	`a2 = foreach a1 generate origin as origin, dest as destination, first(miles) as miles;`

Aggregate Function	Description

`last()`	Returns the value for the last tuple.
	For example, to compute the distance between each combination of origin and destination:
	`a1 = group a by (origin, dest);`
	`a2 = foreach a1 generate origin as origin, dest as destination, last(miles) as miles;`

`max()`	Returns the maximum value of a field. This function takes only a measure as an argument. It can't take a dimension.

`median()`	Accepts a grouped expression of numeric type and returns the middle number (by sorted order, ignoring null values). If there is no one middle number (in other words, the count of non-null values is even), then median returns the average of the two numbers closest to the middle.
	The expression can be any identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(distance), or a literal, such as 2.5.
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, median(miles) as medMiles;`
	`limit q 5;`
	If median is not preceded by a group by clause, it treats each individual row as its own group:
	`q = load "data/airline";`
	`q = foreach q generate dest, median(miles) as medMiles;`
	`limit q 5;`

`min()`	Returns the minimum value of a field. This function takes only a measure as an argument. It can't take a dimension.

`sum()`	Returns the sum of a numeric field.
	`a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";`
	`a = filter a by dest in ["ORD", "LAX", "ATL", "DFW", "PHX", "DEN", "LGA"];`
	`a = group a by carrier;`
	`b = foreach a generate carrier as airline, sum(miles) as miles;`

`unique()`	Returns the count of unique values.
	For example, to find how many origins and destinations a carrier flies from:
	`a1 = group a by carrier;`
	`a2 = foreach a1 generate carrier as carrier, unique(origin) as origins, unique(dest) as destinations;`
	Note: The best way to add summaries (for example, a summary row on a compare table) using *`unique()`* is to use it as a windowing function.

Aggregate Function	Description

`stddev()`	Returns the sample standard deviation computed on the group.
	Accepts a grouped expression of numeric type. If the number of non-null values in the group is equal to 1, `stddev` return null. Otherwise, `stddev` returns the sample standard deviation computed on the group, ignoring null values.
	The expression can be any numeric identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(price), or a literal, such as 2.5.
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, stddev(miles) as stddevMiles;`
	`limit q 5;`

`stddevp()`	Returns the population standard deviation computed on the group.
	Accepts a grouped expression of numeric type and returns the population standard deviation computed on the group, ignoring null values. The expression can be any numeric identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(price), or a literal, such as 2.5.
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, stddevp(miles) as stddevMiles;`
	`limit q 5;`

`var()`	Returns the sample variance (also called the unbiased variance) computed on the group.
	Accepts a grouped expression of numeric type. If the number of non-null values in the group is equal to 1, `var` return null. Otherwise, `var` returns the sample variance computed on the group, ignoring null values. The expression can be any numeric identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(price), or a literal, such as 2.5.
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, var(miles) as varMiles;`
	`limit q 5;`

`varp()`	Returns the population variance (also called the biased variance) computed on the group.
	Accepts a grouped expression of numeric type and returns the population variance computed on the group, ignoring null values. The expression can be any numeric identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(price), or a literal, such as 2.5.
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, varp(miles) as varMiles;`
	`limit q 5;`

`percentile_disc()`	Computes a specific percentile for sorted values in an entire rowset or within distinct partitions of a rowset. The full syntax is: **percentile_disc**(**p** as ***numeric***) **within group** (**order by expr** [**asc** | **desc**])

Aggregate Function	Description

	The `percentile_disc` function accepts a grouped expression `expr` of numeric type and sorts it in the specified order (`asc` or `desc`). If order is not specified, the default order is `asc`. It returns the value behind which (100*p)% of values in the group would fall in the sorted order, ignoring null values.
	
	`p` can be any real numeric value between 0 and 1, and is accurate to 8 decimal places of precision. `expr` can be any identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(distance), or a literal, such as 2.5.
	
	If `expr` contains no value that falls exactly at the 100*p-th percentile mark, `percentile_disc` will return the next value from `expr` in the sort order.
	
	For example, if `Mea1` contains the values `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]` then:
	
	`percentile_disc(0.25) within group (order by Mea1 asc) = 4`
	`percentile_disc(0.25) within group (order by Mea1 desc) = 9`
	`percentile_disc(0) within group (order by Mea1 asc) = 0`
	`percentile_disc(1) within group (order by Mea1 asc) = 13`
	
	Example query:
	
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, percentile_disc(0.25) within group (order by miles desc) as perMiles;`
	`limit q 5;`

`percentile_cont()`	Calculates a percentile based on a continuous distribution of the column value. The full syntax is:
	
	percentile_cont(**p** as ***numeric***) **within group** (**order by expr** [**asc** | **desc**])
	
	The `percentile_cont` function accepts a grouped expression `expr` of numeric type and sorts it in the specified order (`asc` or `desc`). If the order is not specified, the default order is `asc`. It returns the value behind which (100*p)% of values in the group would fall in the sorted order, ignoring null values.
	
	`p` can be any real numeric value between 0 and 1. `expr` can be any identifier, such as 'xInt' or 'price', but cannot be a complex expression, such as price/100 or ceil(distance), or a literal, such as 2.5.
	
	If `expr` contains no value that falls exactly at the 100*p-th percentile mark, `percentile_cont` returns a value linear interpolated from the two closest values in `expr`.
	
	For example, if `Mea1` contains the values `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]` then:
	
	`percentile_cont(0.25) within group (order by Mea1 asc) = 3.25`
	`percentile_cont(0.25) within group (order by Mea1 desc) = 9.75`
	`percentile_cont(0) within group (order by Mea1 asc) = 0`
	`percentile_cont(1) within group (order by Mea1 asc) = 13`
	
	Example query:
	
	`q = load "data/airline";`
	`q = group q by dest;`
	`q = foreach q generate dest, percentile_cont(0.25) within group (order by miles) as perMiles;`
	`limit q 5;`

Aggregate Function	Description

`regr_slope(y, x)`	The simple linear regression function *`regr_slope`* accepts a grouped dependent numeric expression *`y`* and a grouped independent numeric expression *`x`*, and returns the slope of the regression line. The *`regr_slope`* function only considers pairs of (*`x, y`*) values where both values are not *`null`*, and returns *`null`* if there exists fewer than 2 such pairs in the given group. Simple linear regression functions work like aggregation functions on simple grouped values, but do not work with cogroups.
	Example query:
	`q = load "data/sales";`
	`q = group q by all;`
	`q = foreach q generate regr_slope('profit', 'sales') as slope;`
	`limit q 1;`

`regr_intercept(y, x)`	The simple linear regression function *`regr_intercept`* accepts a grouped dependent numeric expression *`y`* and a grouped independent numeric expression *`x`*, and returns the y-intercept for the regression line. The *`regr_intercept`* function only considers pairs of (*`x, y`*) values where both values are not *`null`*, and returns *`null`* if there exists fewer than 2 such pairs in the given group. Simple linear regression functions work like aggregation functions on simple grouped values, but do not work with cogroups.
	Example query:
	`q = load "data/sales";`
	`q = group q by all;`
	`q = foreach q generate regr_intercept('profit', 'sales') as intercept;`
	`limit q 1;`

`regr_r2(y, x)`	The simple linear regression function *`regr_r2`* accepts a grouped dependent numeric expression *`y`* and a grouped independent numeric expression *`x`*, and returns the coefficient of determination (also called R-squared or goodness of fit) for the regression. The *`regr_r2`* function only considers pairs of (*`x, y`*) values where both values are not *`null`*, and returns *`null`* if there exists fewer than 2 such pairs in the given group. Simple linear regression functions work like aggregation functions on simple grouped values, but do not work with cogroups.
	Example query:
	`q = load "data/sales";`
	`q = group q by all;`
	`q = foreach q generate regr_r2('profit', 'sales') as r2;`
	`limit q 1;`

Date Functions

To specify dates in a SAQL query, use date functions and relative date keywords.

Note: Relative dates are relative to UTC, not local time. Data returned for relative dates reflect dates based on UTC time, which may be offset from your local time.

Functions

This table lists SAQL date functions:

Date Function	Description
`date(`***`year, month, day`***`)`	Returns a date. Specify 3 dimensions of a date in the following order: year, month, day. For example: `date('OrderDate_Year', 'OrderDate_Month', 'OrderDate_Day')`
`date_diff(`***`datepart,startdate,enddate`***`)`	Returns an integer representing the interval that has elapsed between two dates. *`datepart`* indicates the interval part to calculate: • year • month • quarter • day • week • hour • minute • second *`startdate`* indicates the start date. *`enddate`* indicates the end date. The difference between two dates is calculated based on the difference in the indicated date part. For example, the year difference between two dates is calculated by subtracting the year part of *`startdate`* from the year part of *`enddate`*. `date_diff("year", toDate("31-12-2015", "dd-MM-yyyy"), toDate("1-1-2016", "dd-MM-yyyy"))` would give a result of 1. Similarly, using the date part month as an example: `date_diff("month", toDate("31-12-2015", "dd-MM-yyyy"), toDate("1-1-2016", "dd-MM-yyyy"))` would also give a result of 1. If *`startdate`* is after *`enddate`* the result is a negative integer of the difference. Examples: **Query** / **Result** `date_diff("year", '2004-02-29', '2005-02-28')` / 1

Date Function	Description	
	`date_diff("year", '2012-1-1', '2012-12-31')`	0
	`date_diff("month", '2003-02-01', '2003-05-01')`	3
	`date_diff("month", '2004-02-28', '2004-03-31')`	1
	`date_diff("quarter", '2012-12-12', '2013-01-05')`	1
	`date_diff("week", '2012-12-12', '2013-01-05')`	3
	`date_diff("day", '2012-12-12', '2013-01-05')`	24
	`date_diff("hour", '2012-12-12', '2013-01-05')`	576
	`date_diff("minute", '2012-12-12', '2013-01-05')`	34560
	`date_diff("second", '2016-09-15 19:42:36', '2016-09-16 19:42:36')`	86400

Query example:

```
q = load \"em/dates\";
q = foreach q generate date_diff("year",
toDate(DateOfBirth, "yyyy-MM-dd"), now()) as age;
q = order q by age asc;
```

Date Function	Description

	Invalid examples:
	`q = group q by date_diff("month", toDate(DateOfBirth), "yyyy-MM-dd"), toDate(RegisteredDate));`
	`q = order q by date_diff("year", toDate(DateOfBirth), "yyyy-MM-dd"), toDate(RegisteredDate));`
	`q = filter q by date_diff("day", toDate(DateOfBirth), "yyyy-MM-dd"), now());`

`dateRange(`***`startArray_y_m_d`***`,` ***`endArray_y_m_d`***`)`	Returns a fixed date range. The first parameter is an array that specifies the start date in the range. The second parameter is an array that specifies the end of the range. For example: `dateRange([1970, 1, 1], [1970, 1, 31])`
`day_in_month(`***`date`***`)`	Returns an integer representing the day of the month for a specific date. See `day_in_week` for usage.
`day_in_quarter(`***`date`***`)`	Returns an integer representing the day of the quarter for a specific date. See `day_in_week` for usage.
`day_in_week(`***`date`***`)`	Returns an integer representing the day of the week for a specific date. 1 = Sunday, 2 = Monday and so on. *`date`* indicates the reference date. Example: `q = foreach q generate day_in_week(toDate(OrderDate));`
`day_in_year(`***`date`***`)`	Returns an integer representing the day of the year for a specific date. See `day_in_week` for usage.

`daysBetween(`***`date1, date2`***`)`	Returns the number of days between 2 dates as an integer.
	The `daysBetween()` function can't take dimensions as arguments directly. Pass `toDate()` and `now()` functions as arguments.
	`q = foreach q generate daysBetween(toDate(OrderDate, "yyyy-MM-dd"), now()) as daysToShip;`
	`q = foreach q generate daysBetween(toDate(OrderDate, "yyyy-MM-dd"), toDate(ShipDate, "yyyy-MM-dd")) as daysToShip;`
	`q = foreach q generate daysBetween(toDate(OrderDate_Year + ":" + OrderDate_Month + ":" + OrderDate_Day, "yyyy:MM:dd"), toDate(ShipDate_Year + ":"`

<table>
<tr><th>Date Function</th><th>Description</th></tr>
<tr><td></td><td>

```
   + ShipDate_Month + ":" + ShipDate_Day, "yyyy:MM:dd"
)) as daysToShip;
```

</td></tr>
<tr><td>month_days(date)</td><td>Returns the number of days in the month for a specific date.

date indicates the reference date.

Examples:

<table>
<tr><th>Query</th><th>Result</th></tr>
<tr><td>month_days(toDate('2004-02-12', "yyyy-MM-dd")</td><td>29</td></tr>
<tr><td>month_days(toDate('2012-04-07', "yyyy-MM-dd")</td><td>30</td></tr>
<tr><td>month_days(toDate('1990-13-11', "yyyy-MM-dd")</td><td>NULL</td></tr>
</table>

Query example:

```
q = load \"em/dates\";
q = foreach q generate month_days(toDate(BillDate,
"yyyy-MM-dd")) as BillingMonth;
q = order q by BillingMonth asc;
```

Invalid examples:

```
q = group q by month_days(toDate(BillDate,
"yyyy-MM-dd"));
```

```
q = order q by month_days(toDate(BillDate,
"yyyy-MM-dd"));
```

```
q = filter q by month_days(toDate(BillDate,
"yyyy-MM-dd"));
```

</td></tr>
<tr><td>month_last_day(date)</td><td>Returns the date of the last day of the month for a specific date. See week_last_day for usage.</td></tr>
<tr><td>now()</td><td>Returns current datetime in UTC. This function is valid in a foreach statement only.

```
q = foreach q generate now() as now;
```

This function is commonly used in daysBetween() and toString() functions.</td></tr>
<tr><td>quarter_days(date)s</td><td>Returns the number of days in the quarter for a specific date. See month_days for usage.</td></tr>
</table>

Date Function	Description
`quarter_last_day(`***`date`***`)`	Returns the date of the last day of the quarter for a specific date. See `week_last_day` for usage.

`toDate(`***`string [,formatString`***`])`	Converts a string to a date. If a *`formatString`* argument isn't provided, the function uses the format `yyyy-MM-dd HH:mm:ss`.
	`q = foreach q generate toDate(OrderDate);`
	`q = foreach q generate toDate(OrderDate_Day + \"-\" + OrderDate_Month + \"-\" + OrderDate_Year, \"dd-MM-yyyy\");`
	This function is often passed as an argument to `daysBetween()` or `toString()`.

`toDate(`***`epoch_seconds`***`)`	Converts Unix epoch seconds to a date. If `epoch_seconds` is `0`, `toDate(epoch_seconds)` returns `'1970-01-01 00:00:00'`. This function is convenient for adding or subtracting time periods to or from a date. When adjusting dates for time zone differences, adding or subtracting the number of seconds in the time difference produces the correct local date. If the time crosses the local meridian, a different date is produced. For example, assuming `Current_Date` is the current date expressed as the number of seconds since `'1970-01-01 00:00:00'`, then the function `toDate(Current_Date - 8*3600)` subtracts 8 hours. Refer to Working with Time Zones for a practical example.
`toString(`***`date, formatString`***`)`	Converts a date to a string. This function must take a `toDate()` or `now()` function as its first argument. `q = foreach q generate toString(now(), \"yyyy-MM-dd HH:mm:ss\") as ds1;`
`week_last_day(`***`date`***`)`	Returns the date of the last day of the week for a specific date. *`date`* indicates the reference date. Examples: **Query** — **Result** week_last_day(toDate('2016-12-08', "yyyy-MM-dd")) — 2016-12-10 week_last_day(toDate('2015-07-05', "yyyy-MM-dd")) — 2015-07-11 week_last_day(toDate('2012-11-33', "yyyy-MM-dd")) — Error

Date Function	Description

	Query example:
	`q = load \"em/dates\";`
	`q = foreach q generate week_last_day(toDate(BillDate, "yyyy-MM-dd")) as BillingWeek;`
	`q = order q by BillingWeek asc;`
	Invalid examples:
	`q = group q by week_last_day(toDate(BillDate, "yyyy-MM-dd"));`
	`q = order q by week_last_day(toDate(BillDate, "yyyy-MM-dd"));`
	`q = filter q by week_last_day(toDate(BillDate, "yyyy-MM-dd"));`

`year_days(date)`	Returns the number of days in the year for a specific date. See `month_days` for usage.
`year_last_day(date)`	Returns the date of the last day of the year for a specific date. See `week_last_day` for usage. Note: While it's apparent that this function will always return 31st December, it is included for uses such as finding the number of days to the year end, and for use in a specific locale.

Specify Fixed Date Ranges

To specify a range for fixed dates, use the `dateRange()` function. Specify the dates in the order: year, month, day.

Example:

```
a = filter a by date('year', 'month', 'day') in [dateRange([1970, 1, 1], [1970, 1, 11])];
```

Specify Relative Date Ranges

To specify a relative date range, use the `in` operator on an array with relative date keywords. Here are 4 examples:

```
a = filter a by date('year', 'month', 'day') in ["1 year ago".."current year"];
a = filter a by date('year', 'month', 'day') in ["2 quarters ago".."2 quarters ahead"];
a = filter a by date('year', 'month', 'day') in ["4 months ago".."1 year ahead"];
a = filter a by date('year', 'month', 'day') in ["2 fiscal_years ago".."current day"];
```

The relative date keywords are:

- current day
- n day(s) ago
- n day(s) ahead

- current week
- n week(s) ago
- n week(s) ahead
- current month
- n month(s) ago
- n month(s) ahead
- current quarter
- n quarter(s) ago
- n quarter(s) ahead
- current fiscal_quarter
- n fiscal_quarter(s) ago
- n fiscal_quarter(s) ahead
- current year
- n year(s) ago
- n year(s) ahead
- current fiscal_year
- n fiscal_year(s) ago
- n fiscal_year(s) ahead

This table shows the time windows for some of the relative date keywords. In these time window examples, the current day is **2014/12/16** and **FiscalMonthOffset 1** (the fiscal year starts on February 1).

Relative Date Keyword	Start Date	End Date
current day	2014/12/16 00:00:00	2014/12/16 23:59:59
current quarter	2014/10/1 00:00:00	2014/12/31 23:59:59
1 year ago	2013/1/1 00:00:00	2013/12/31 23:59:59
1 month ahead	2015/1/1 00:00:00	2015/1/31 23:59:59
current fiscal_year	2014/2/1 00:00:00	2015/1/31 23:59:59
current fiscal_quarter	2014/11/1 00:00:00	2015/1/31 23:59:59
2 fiscal_quarters ahead	2015/5/1 00:00:00	2015/7/31 23:59:59
current day - 1 year	2013/12/16 00:00:00	2013/12/16 23:59:59
current fiscal_year + 5 days	2014/2/6 00:00:00	2014/2/6 23:59:59

Note: Only standard fiscal periods are supported. See “About Fiscal Years" in Salesforce Help.

Add and Subtract Dates

You can add and subtract dates using the relative date keywords.

Example: Here are examples of time windows for relative date keywords using addition and subtraction. In these time window examples, the current day is **2014/12/16** and **FiscalMonthOffset 1** (the fiscal year starts on February 1).

In this query, the start date is **2013-12-16 00:00:00** and the end date is open ended:

```
a= filter a by date('year', 'month', 'day') in ["current day - 1 year"..] ;
```

In this query, the start date is **2014-12-16 00:00:00** and the end date is **2017-3-31 23:59:59**:

```
a= filter a by date('year', 'month', 'day') in ["current day".."2 years ahead + 3
months"];
```

Here's how to determine the end date: the year is 2014, so 2 years ahead is 2016, which has a year end time of 2016-12-31 23:59:59. When you add 3 months, the total end date is 2017-3-31 23:59:59.

In this query, the start date is **2014-2-6 00:00:00** and the end date is **2017-3-31 23:59:59**:

```
a= filter a by date('year', 'month', 'day') in ["current fiscal_year + 5 days".."2
years ahead + 3 months"];
```

Use Open-Ended Relative Date Ranges

To build queries like "List all opportunities closed after 12/23/2014" and "Get a list of marketing campaigns from before 04/2/2015," use open-ended date ranges.

Example: This example shows an open-ended relative date range.

```
a = filter a by date('year','month','day') in [.."current month"];
```

Example: This example shows an open-ended fixed date range. The date format of `OrderDate` is `yyyy-MM-dd`.

```
q = filter q by OrderDate in ["2015-01-01"..];
```

Working with Time Zones

A practical use of the `toDate()` function is to calculate time zone changes for an Analytics dashboard. This JSON code fragment uses a `computeExpression` action in a transformation, which in turn uses a `saqlExpression` to call the `toDate()` function. This technique enables a dashboard to show the most appropriate time and date, whether local or UTC.

```
"Extract_Opportunity": {
  "action": "computeExpression",
  "parameters": {
    "source": "Digest_Opportunity",
    "mergeWithSource": true,
    "computedFields": [
     {
      "name": "CreatedDateNew",
      "type": "Date",
      "format": "MM/dd/yyyy",
      "saqlExpression": "toDate(CreatedDate_sec_epoch - 8*3600)"
     }
    ]
  }
},
```

The example takes an existing date `CreatedDate_sec_epoch` and subtracts 8 hours to create a new date `CreateDateNew`. The table shows how the calculation changes the (formatted) `CreatedDateNew` dates. In each case, the time change has also changed the date.

CreatedDate_sec_epoch	CreatedDateNew
2015-11-03T06:49:25.00OZ	11/2/2015
2014-08-19T06:42:33.00OZ	8/18/2014
2014-09-28T03:12:25.00OZ	9/27/2014

Refer to the computeExpression topic for further information.

String Functions

To perform string operations in a SAQL query, use string functions.

While SAQL operators support strings, and the `coalesce()` function returns the first non-null item in a list including strings, the following table lists SAQL functions specifically for manipulating strings.

Functions

This table lists the SAQL string functions:

Function	Description

`ends_with(`*`string`*`, `*`suffix`*`)`	This function returns *`true`* if *`string`* ends with *`suffix`*, and *`false`* otherwise. String comparison is case-sensitive. If any of the parameters are *`null`*, then the function returns *`null`*. If *`suffix`* is an empty string, then the function returns *`null`*.
	`ends_with("FIT", "T") == true`
	`ends_with("FIT", "BIT") == false`

`index_of(`*`string`*`, `*`searchStr`*` [,`*`position`*` [, `*`occurence`*`]])`	This function returns the index of the specified occurrence of *`searchStr`* in *`string`* beginning at the specified *`position`*. The function returns 0 if *`searchStr`* is not found. This function is case-sensitive. If any of the parameters are *`null`*, then the function returns *`null`*.
	The default value of *`position`* is 1, which means that the function begins searching at the first character of *`string`*. An error results if *`position`* is negative or zero.
	If present, *`occurrence`* is an integer indicating which occurrence within *`string`* to search for. The value of occurrence must be positive, and defaults to 1 if omitted. So for example, if there is more than one matching occurrence, and *`occurence`* is 2, the index of the second occurrence is returned.
	Constant values are supported for *`position`* and *`occurrence`*, not arbitrary expressions.
	If *`searchStr`* is an empty string, then the function returns *`null`*.
	`index_of("Hawaii", "a") == 2`
	`index_of("Hawaii", "a", 2) == 2`

Function	Description

	`index_of("Hawaii", "a", 3) == 4`
	`index_of("Hawaii", "a", 3, 2) == 0`
	`index_of("Hawaii", "i", -1, 1) == error`
	`index_of("Hawaii", "i", -3, 1) == error`
	`index_of("", "i") == null`
	`index_of("i", "") == null`

`len(`*`string`*`)`	This function returns the number of characters in the string.
	`len` returns the length of *`string`* in characters. If *`string`* is *`null`*, then `len(`*`string`*`)` is also *`null`*.
	Leading and trailing whitespace characters are included in the length returned.
	`len("starfox") == 7`
	`len(" rocket ") == 8`
	`len("□") == 1`
	`len("") == 0`

`lower(`*`string`*`)`	This function returns *`string`* with all characters in lowercase. If *`string`* is *`null`*, then the result is *`null`*. Refer to the note for `upper()` concerning Unicode case mapping. `lower("JAVA") == "java"`

`ltrim(`*`string`*`,`*`chars`*`)`	This function removes the left part of a string up to the specified characters, or removes leading spaces.
	`ltrim` returns the value of *`string`* with the initial characters removed up to the first character not in *`chars`*.
	`chars` may contain multiple characters. If *`chars`* is omitted, leading space characters are removed. If *`string`* or *`chars`* is *`null`*, then the result is *`null`*.
	`ltrim("__c__val__", "_") == "c__val__"`
	`ltrim(string, " \t\r") == ltrim(string)`
	`ltrim("aabcd", "ab") == "cd"`

`replace(`*`string, searchStr, replaceStr`*`)`	This function returns *`string`* with every occurrence of *`searchStr`* replaced by *`replaceStr`*. If any of the parameters are *`null`*, then the function returns *`null`*. If *`searchStr`* is an empty string, *`null`* is returned. This function is case-sensitive.
	`replace("Watson, come quickly.", "quickly", "slowly") == "Watson, come slowly."`
	`replace("Watson, come quickly.", "o", "a") == "Watsan, came quickly."`
	`replace("Watson, come quickly.", "", "Mr.") == null`

`rtrim(`*`string`*`,`*`chars`*`)`	This function removes the right part of a string back to the specified characters, or removes trailing spaces. `rtrim` returns the value of *`string`* with the final characters removed back to the first character not in *`chars`*.

Function	Description

	chars may contain multiple characters. If *chars* is omitted, trailing space characters are removed. If *string* or *chars* is *null*, then the result is *null*.
	`rtrim("__c__val__", "_") == "__c__val"`
	`rtrim(ltrim(string, " \t\r"), " \t\r") == trim(string, " \t\r")`

starts_with(*string*, *prefix*)	This function returns *true* if *string* starts with *prefix*, and *false* otherwise. String comparison is case-sensitive. If any of the parameters are *null*, then the function returns *null*. If *prefix* is an empty string, then the function returns *null*.
	`starts_with("FIT", "F") == true`
	`starts_with("FIT", "BIT") == false`

substr(*string*,*position*[, *length*])	This function returns a substring starting at a specified position and, optionally, of the specified length.
	substr returns *length* characters of *string*, beginning at character position *position*. If *length* is omitted, then *length* = len(*string*), so all characters are returned from *position* to the end of the string. If any of the parameters are *null*, then the function returns *null*.
	The first character in *string* is at position 1. If *position* is negative then the position is relative to the end of the string. So a *position* of -1 denotes the last character.
	If *length* is negative, then the function returns *null*. If *position* > len (*string*) or *position* < -len(*string*) or *position* = 0, then the empty string is returned.
	`substr("CRM", 1, 1) == "C"`
	`substr("CRM", 1, 2) == "CR"`
	`substr("CRM", -1, 1) == "M"`
	`substr("CRM", -2, 2) == "RM"`
	`substr("CRM", 4, 1) == ""`

trim(*string*,*chars*)	This function removes the left and right part of a string up to the specified characters, or removes leading and trailing spaces.
	trim returns the value of *string* with the initial and final characters removed to the first character not in *chars*.
	chars may contain multiple characters. If *chars* is omitted, leading and trailing space characters are removed. If *string* or *chars* is *null*, then the result is *null*.
	`trim("__c__val__", "_") == "c__val"`
	`trim("__c__val__", "_c") == "val"`
	`trim("  c__val  ") == "c__val"`
	`trim("  c__val  ") == ltrim(rtrim("  c__val  "))`
	`trim("aaaaaa", "a") == ""`

Function	Description

`upper(string)`	This function returns `string` with all characters in uppercase. If `string` is `null`, then the result is `null`.
	`upper("go") == "GO"`
	`upper ("große") == "GROßE"`
	Note: The behavior of the `upper()` and `lower()` functions—and the characters affected by them—is determined by the default case mapping of the Unicode standard. The mapping considers each Unicode character in isolation without regard for context or language-specific rules. The example above does not reflect the German language handling of the ß character. A natural-language conversion would produce GROSSE.

Math Functions

To perform numeric operations in a SAQL query, use math functions.

You can use SAQL math functions in `foreach` statements and in the `filter by` clause after a `foreach` statement.

You can't use math functions in a `group by` clause or in an `order by` clause. You also can't use math functions in the `filter by` clause before a `foreach` statement, but you **can** use them after the `foreach` statement.

Functions

This table lists the SAQL math functions:

Function	Description

`abs(n)`	Returns the absolute number of `n` as a numeric value. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308.
	This example is valid:
	`q = foreach q generate abs(pct_change) as pct_magnitude;`
	These examples are invalid:
	`q = group q by abs(pct_change);`
	`q = order q by abs(pct_change);`

`ceil(n)`	Returns the nearest integer of equal or greater value to `n`. `n` can be any real numeric value in the range of -1e308 <= n <= 1e308.
	This example is valid:
	`q = foreach q generate ceil(miles) as distance;`
	These examples are invalid:
	`q = group q by ceil(miles);`
	`q = order q by ceil(miles);`

Function	Description

`floor(n)`	Returns the nearest integer of equal or lesser value to `n`. `n` can be any real numeric value in the range of -1e308 <= n <= 1e308.
	This example is valid:
	`q = foreach q generate floor(miles) as distance;`
	These examples are invalid:
	`q = group q by floor(miles);`
	`q = order q by floor(miles);`

`trunc(n[, m])`	Returns the value of the numeric expression `n` truncated to `m` decimal places. `m` can be negative, in which case the function returns `n` truncated to -`m` places to the left of the decimal point. If `m` is omitted, it returns `n` truncated to the integer place. `n` can be any real numeric value in the range of -1e308 <= n <= 1e308. `m` can be an integer value between -15 and 15 inclusive.
	This example is valid:
	`q = foreach q generate trunc(Price, 2) as Price;`
	These examples are invalid:
	`q = group q by trunc(Price, 2);`
	`q = order q by trunc(Price, 2);`

`round(n[, m])`	Returns the value of `n` rounded to `m` decimal places. `m` can be negative, in which case the function returns `n` rounded to -`m` places to the left of the decimal point. If `m` is omitted, it returns `n` rounded to the nearest integer. For tie-breaking, it follows round half way from zero convention. `n` can be any real numeric value in the range of -1e308 <= n <= 1e308. `m` can be an integer value between -15 and 15, inclusive.
	This example is valid:
	`q = foreach q generate round(Price, 2) as Price;`
	These examples are invalid:
	`q = group q by round(Price, 2);`
	`q = order q by round(Price, 2);`

`exp(n)`	Returns the value of Euler's number `e` raised to the power of `n`, where `e` = 2.71828183… The smallest value for `n` that will not result in 0 is 3e-324. `n` can be any real numeric value in the range of -1e308 <= n <= 700.
	These examples are valid:
	`q = foreach q generate exp(value) as value;`
	`q = filter q by exp(value) < 5;`
	These examples are invalid:
	`q = group q by exp(value);`
	`q = order q by exp(value);`

Function	Description

`log(m, n)`	Returns the natural logarithm (base m) of a number *n*. The values *m* and *n* can be any positive, non-zero numeric value in the range 0 < m, n <= 1e308 and m ≠ 1.
	The smallest number input allowed for *m* is >0, m!=1. The smallest number for *m* or *n* that will not produce 0 is log(10, 0.3e-323).
	These examples are valid:
	`q = foreach q generate log(10, Population) as Population;`
	`q = filter q by log(10, Population) < 15;`
	These examples are invalid:
	`q = group q by log(10, Population);`
	`q = order q by log(10, Population);`

`power(m, n)`	Returns *m* raised to the *n*th power. *m*, *n* can be any numeric value in the range of -1e308 <= *m*, *n* <= 1e308. Returns null if *m* = 0 and *n* < 0.
	• If *m* = 0, *n* must be a non-negative value.
	• If *m* < 0, *n* must be an integer value.
	• The result of power(*m*, *n*) must be within the range expressed by a float64 number.
	These examples are valid:
	`q = foreach q generate power(length, 2) as area, length;`
	`q = filter q by power(length, 2) > 10;`
	These examples are invalid:
	`q = group q by power(length, 2);`
	`q = order q by power(length, 2);`

`sqrt(n)`	Returns the square root of a number *n*. The value *n* can be any non-negative numeric value in the range of 0 <= *n* <= 1e308.
	These examples are valid:
	`q = foreach q generate sqrt(value) as value;`
	`q = filter q by sqrt(value) < 10;`
	These examples are invalid:
	`q = group q by sqrt(value);`
	`q = order q by sqrt(value);`

Windowing Functions

Use SAQL windowing functionality to calculate common business cases such as percent of grand total, moving average, year and quarter growth, and ranking.

SAQL now supports windowing, using a syntax inspired by SQL. Windowing functions allow you to calculate data for a single group using aggregated data from adjacent groups. Windowing does not change the number of rows returned by the query. Windowing aggregates across groups rather than within groups and accepts any valid numerical projection on which to aggregate.

Windowing with an aggregate function uses the following syntax:

```
<windowfunction>(<projection expression>) over (<row range> partition by <reset groups>
order by <order clause>) as <label>
```

When using ranking functions, use the following syntax:

```
<rankfunction> over([..] partition by <reset groups> order by <order clause>) as <label>
```

Where:

windowfunction
An aggregate function that supports windowing. Currently supported functions are `avg`, `sum`, `min`, `max`, `count`, `median`, `percentile_disc`, and `percentile_cont`.

rankfunction
Returns a rank value for each row in a partition. The following ranking functions are supported: `rank()`, `dense_rank()`, `cume_dist()` and `row_number()`. Refer to the Ranking Functions section for examples.

projection expression

The expression used to generate a projection from the values of specified columns.

row range

Row ranges are specified using the following syntax.

Range	Meaning
[.. 0]	From beginning to current row in the reset group.
[0 ..]	From current row to the last row in the reset group.
[-2 .. 0]	From two rows prior to current row. Window covers 3 rows.
[0 .. 2]	From current row to 2 rows ahead of current row. Windows covers 3 rows.
[-1 .. -1]	One row prior to current row. Window includes a single row.
[.. -2]	From beginning of reset group to 2 rows prior to current row.
[..]	Aggregates the entire reset group.

reset groups
The column(s) which reset windowing aggregation when their value(s) change. A reset group of `all` indicates no reset boundaries for the window aggregation.

order clause
Specify column(s) by which to sort. This orders the rows before the window function gets evaluated.

Note: The order clause is not allowed on expressions where the row range is `[..]` and the window function is `sum`, `avg`, `min`, or `max`. For example, `sum(sum(Sales)) over([..] partition by Year order by Quarter)` is invalid.

label
The output column name.

Notes

Grouped Queries

Windowing functionality is enabled only for grouped queries. The following is **not** valid:

```
a = load "dataset";
b = foreach a generate sum(sum(sales)) over([.. 0] partition by all order by all);
```

Multiple Resets and Multiple Orders

Multiple resets and multiple orders are valid. For example:

```
sum(sum(Sales)) over([-2 .. 0] partition by (OrderDate_Year, OrderDate_Quarter) order
by OrderDate_Year)

sum(sum(Sales)) over([-2 .. 0] partition by (Year, Quarter) order by (Year asc, sum(Sales)
 desc))
```

Cogroups

Windowing functions can be used with cogroup queries. For example:

```
sum(sum(a[Sales])) over([-2 .. 0] partition by (a[Year], a[Quarter]) order by (a[Year]
asc, sum(a[Sales]) desc))
```

Note: Each Windowing function can be used with only 1 cogroup stream. The following is **not** valid:

```
a = load "dataset1";
b = load "dataset2";
c = group a by column1, b by column2;
d = foreach c generate sum(sum(a[sales])) over([.. 0] partition by b[column2] order
 by all)
```

Refer to the Aggregate Functions topic for details on function usage.

Examples

Running Total (No Reset)

The following query calculates the running total of sum of sales every quarter, with "partition by all" denoting that the sum is not reset by any column.

```
q = load "dataset";
q = group q by (OrderDate_Year, OrderDate_Quarter);
q = foreach q generate OrderDate_Year as Year, OrderDate_Quarter as Quarter, sum(Sales)
as sum_amt, sum(sum(Sales)) over([.. 0] partition by all order by (OrderDate_Year,
OrderDate_Quarter)) as r_sum;
```

Year	Quarter	sum_amt	r_sum
2013	1	1000	1000

Year	Quarter	sum_amt	r_sum
2013	2	2000	3000
2013	3	3000	6000
2013	4	2000	8000
2014	1	1000	9000
2014	2	500	9500
2014	3	9000	18500
2014	4	3000	21500
2015	1	500	22000
2015	2	500	22500
2015	3	200	22700
2015	4	400	23100

Running Totals By Year

Running total resets on every year.

```
q = load "dataset";
q = group q by (OrderDate_Year, OrderDate_Quarter);
q = foreach q generate OrderDate_Year as Year, OrderDate_Quarter as Quarter, sum(Sales)
as sum_amt, sum(sum(Sales)) over([.. 0] partition by OrderDate_Year order by (OrderDate_Year,
 OrderDate_Quarter)) as r_sum;
```

Year	Quarter	sum_amt	r_sum
2013	1	1000	1000
2013	2	2000	3000
2013	3	3000	6000
2013	4	2000	**8000**
2014	1	1000	1000
2014	2	500	1500
2014	3	9000	10500
2014	4	3000	**13500**
2015	1	500	500
2015	2	500	100
2015	3	200	1200
2015	4	400	**1600**

Min Sales Trailing 3 Quarters (Moving Min)

Finds the moving minimum values in the window of last two rows to current row.

```
q = load "dataset";
q = group q by (OrderDate_Year, OrderDate_Quarter);
q = foreach q generate OrderDate_Year as Year, OrderDate_Quarter as Quarter, sum(Sales)
as sumSales, min(sum(Sales)) over([-2 .. 0] partition by OrderDate_Year order by
(OrderDate_Year, OrderDate_Quarter)) as m_min;
```

Year	Quarter	sumSales	m_min
2013	1	1000	1000
2013	2	2000	1000
2013	3	3000	1000
2013	4	2000	**2000**
2014	1	1000	1000
2014	2	500	500
2014	3	9000	500
2014	4	3000	**500**
2015	1	4000	4000
2015	2	500	500
2015	3	200	200
2015	4	400	200

Percentage Total

This query calculates the percentage of the quarter's sales for the year. Row range [..] calculates the subtotals of each year, which is used in the formula to calculate the percentage.

```
q = load "dataset";
q = group q by (OrderDate_Year, OrderDate_Quarter);
q = foreach q generate OrderDate_Year as Year, OrderDate_Quarter as Quarter, sum(Sales)
as sumSales, (sum(Sales) * 100) / sum(sum(Sales)) over([..] partition by OrderDate_Year)
as p_tot;
```

Year	Quarter	sumSales	p_tot
2013	1	1000	**12.5%**
2013	2	2000	**25%**
2013	3	3000	**37.5%**
2013	4	2000	**25%**
2014	1	1000	7.41%

Year	Quarter	sumSales	p_tot
2014	2	500	3.70%
2014	3	9000	66.67%
2014	4	3000	22.22%
2015	1	500	**31.25%**
2015	2	500	**31.25%**
2015	3	200	**12.50%**
2015	4	400	**25%**

Differences Along Year

This query calculates the growth of sales compared with the previous quarter, with [-1 .. -1] referring to the quarter before the quarter on the row. The blank spaces in the result table represent null values.

```
q = load "dataset";
q = group q by (OrderDate_Year, OrderDate_Quarter);
q = foreach q generate OrderDate_Year as Year, OrderDate_Quarter as Quarter, sum(Sales)
as sumSales, sum(Sales) - sum(sum(Sales)) over([-1 .. -1] partition by OrderDate_Year order
 by (OrderDate_Year, OrderDate_Quarter)) as diff;
```

Year	Quarter	sumSales	diff
2013	1	1000	
2013	2	2000	1000
2013	3	3000	1000
2013	4	2000	-1000
2014	1	1000	
2014	2	500	-500
2014	3	9000	8500
2014	4	3000	-6000
2015	1	500	
2015	2	500	0
2015	3	200	-300
2015	4	400	200

Ranking Functions

rank()

Assigns rank based on order. Repeats rank when the value is the same, and skips as many on the next non-match.

dense_rank()
Same as rank() but doesn't skip values on previous repetitions.

cume_dist()
Calculates the cumulative distribution (relative position) of the data in the reset group.

row_number()
Assigns a number incremented by 1 for every row in the reset group.

Examples

```
q = load "dataset";
q = group q by (Year, Quarter);
q = foreach q generate Year, Quarter, sum(Sales) as sum_amt, rank() over([..] partition
by Year order by sum(Sales)) as rank;
```

The following table also shows result columns as if the `dense_rank()`, `cume_dist()` and `row_number()` functions were substituted for `rank()` in the previous code.

Year	Quarter	sum_amt	rank	dense_rank	cume_dist	row_number
2013	1	1000	1	1	0.25	1
2013	2	2000	2	2	0.75	2
2013	4	2000	2	2	0.75	3
2013	3	3000	4	3	1	4
2014	2	500	1	1	0.25	1
2014	1	1000	2	2	0.5	2
2014	4	3000	3	3	0.75	3
2014	3	9000	4	4	1	4
2015	1	500	1	1	0.5	1
2015	2	500	1	1	0.5	2
2015	4	600	3	2	0.75	3
2015	3	700	4	3	1	4

This query shows the top 3 performing quarters in a year.

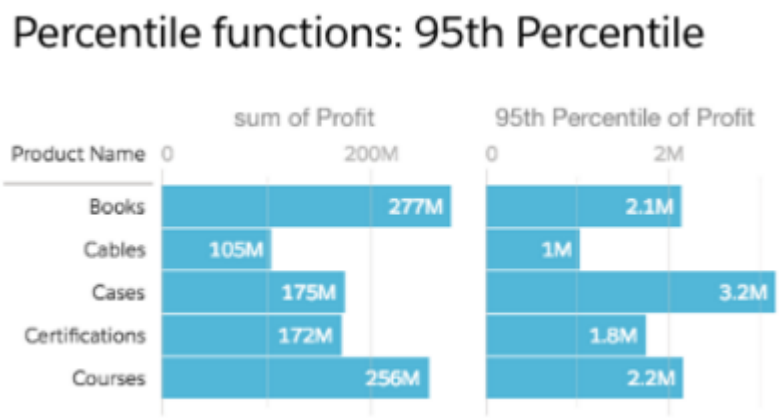
```
q = load "dataset";
q = group q by (Year, Quarter);
q = foreach q generate Year, Quarter, sum(Sales) as sum_amt, rank() over([..] partition
by Year order by sum(Sales)) as rank;
q = filter q by rank <= 3;
```

Year	Quarter	sumSales	rank
2013	1	1000	1
2013	2	2000	2

Year	Quarter	sumSales	rank
2013	4	2000	2
2014	2	500	1
2014	1	1000	2
2014	4	3000	3
2015	1	500	1
2015	2	600	1
2015	4	600	3

This query shows the 95th percentile.

```
q = load "Oppty_Products_Scored";
q = group q by (ProductName);
q = foreach q generate ProductName, sum(TotalPrice) as sum_Price, percentile_cont(0.95)
within group (order by 'TotalPrice') as 'sum_95Percentile';
q = limit q 5;
```



Refer to the Aggregate Functions topic for details on function usage.

coalesce()

Use the `coalesce()` function to get the first non-null value from a list of parameters.

```
coalesce(value1 , value2 , value3 , ... )
```

For example, the following statements ensure that a non-null grouping value is used when doing a full outer join.

```
accounts = load "em/cogroup/accounts";
opps = load "em/cogroup/opportunities";
c = cogroup accounts by 'Year' full, opps by 'Year';
c = foreach c generate coalesce(accounts::'Year',opps::'Year') as 'Group';
```

You can also use the `coalesce()` function to replace nulls with a default value. For example, the following statements set the default for division by zero to a non-null value.

```
q = load "dataset";
q = group q by 'Year';
q = foreach q generate 'Year', coalesce(sum(Amount)/sum(Quantity),0) as 'AvgPrice';
```

QUERY PERFORMANCE

To optimize performance, learn how to structure your query to take advantage of the different stages a SAQL query passed through.

What Happens Behind the Scenes

SAQL stands for Salesforce Analytics Query Language. Most actions you take in Analytics results in one or more SAQL queries. Every lens, dashboard, and explorer action generates and executes a SAQL statement to build the data needed for the visualization.

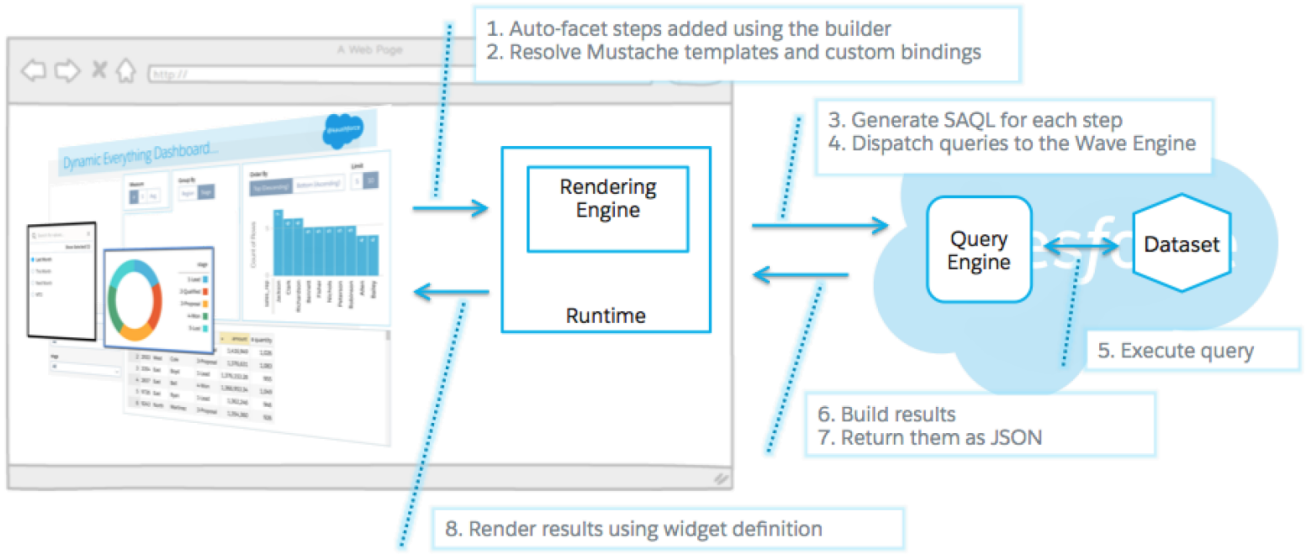
Tip: SAQL is influenced by the Apache Pig-Latin (pigql) syntax, but their implementations differ, and they are not compatible.

When Analytics evaluates the steps, widgets, and layouts to render a dashboard, it does the following:

- It auto-facets the compact steps. In other words, it links different widgets that relate to each other.
- It resolves bindings and templates.
- Every step is converted to a SAQL query.

The SAQL query is then sent to the query engine for execution. The resulting data is passed to the charting library, which renders it using corresponding widget definitions.

How the components fit together



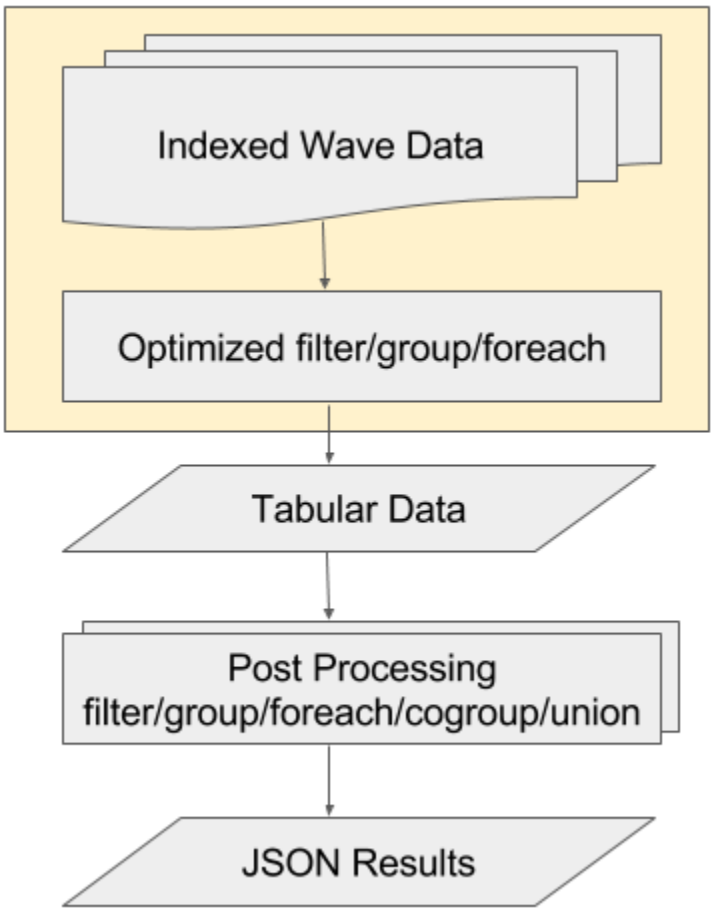
Developers can write SAQL to access Analytics data, either via the Analytics REST API, or by creating/editing SAQL queries contained in the dashboard JSON.

A SAQL query loads an input dataset, operates on it, and outputs a results dataset. Each SAQL statement (a reserved keyword such as filter, group, order, and so on) has an input stream, an operation, and an output stream. Statements can span multiple lines and must end with a semicolon.

Each query line is assigned to a named stream. A named stream can be used as input to any subsequent statement in the same query. The only exception to this rule is the last line in a query, which you don't need to assign explicitly.

A query typically goes through several steps, or layers, on its quest to return the requested data as JSON. The query engine code where this journey starts is designed to execute extremely quickly, and the data is also very efficiently indexed, so query operations (such as filter, group, and foreach) are highly optimized.

The logical layers a query passes through



After that, the data is essentially in tabular form, so any major query operations—filter, (co)group, foreach, or union—are less optimized and require more processing; a more "brute force" approach.

A common use of SAQL is to create derived measures or dimensions. This is a fancy way of saying that you can create new columns using calculations based on existing columns. These calculations may be very simple—perhaps just adding two measures or concatenating two dimensions—or they can be very complex.

The following topics highlight common causes of problematic query design.

Projection is Key

See how changing the order of the functions in your query can give remarkable performance improvements.

Grouping Order

Consider why the order of fields affects how your query is processed.

Network Traffic and Latency
You might not think there's much you can do about network latency, but there are ways to reduce traffic.

Redundant Filters
Is your query doing more work than it needs to? Check to see if you have redundant filters.

Use the ELT Process
Is your dataset set up correctly for what you're trying to do? You may be doing unnessesary work in your queries.

Multi-Value Dimensions
If you use picklists, and find your queries are slow, consider the impact of multi-value dimensions.

Projection is Key

See how changing the order of the functions in your query can give remarkable performance improvements.

Think Projection

With behind-the-scenes knowledge of how data is queried, it quickly becomes apparent that writing queries to take advantage of the super-fast and efficiently indexed layer is key to maximize performance. This before-and-after concept essentially relates to projection.

Tip: What is projection? When a query creates a new stream with a foreach statement—and it's the first foreach in the query—that is a projection.

Pre-projection queries, particularly those dealing with rows numbering in the hundreds of thousands or more, will execute much faster than post-projection queries dealing with the same number of rows as tabular data. So, instead of:

```
q = load "something";
q = foreach q generate 'col1'+'col2' as 'key', col3;
q = filter q by 'key';
q = filter q by 'col3';
q = group q by 'col3';
```

..where the filtering and grouping occur after projection (foreach), change the order so the filtering and grouping occur before projection:

```
q = load "something";
q = filter q by 'col1';
q = filter q by 'col2';
q = filter q by 'col3';
q = group q by 'col3';
q = foreach q generate 'col1'+'col2' as 'key', col3;
```

So a good practice is to ensure that the most demanding part of your query is tackled by the appropriate layer—the layer able to process that filter or grouping most efficiently.

A great many "slow query" cases addressed by support and development teams are ultimately resolved by rewriting the query to perform grouping and filtering before projection.

Note: If you need to filter or group by an expression (e.g. key=col1+col2), the best option for performance is to create the column in the dataset so that it is calculated at ETL time and indexed. See Use the ELT Process.

Grouping Order

Consider why the order of fields affects how your query is processed.

Field Order When Grouping

In addition to optimization through pre-projection queries, another way to improve performance is to carefully consider the order of query fields.

For example, when multiple grouping, consider the cardinality, or number of unique values, for each key field. If the first stage of grouping deals with a very large stream of high cardinality, performance can suffer even with the use of an inverted index. For example, in the following example, Id1 has high cardinality and so takes more processing. For small streams it may not be a big deal, but for very large streams it can affect performance significantly.

```
group q by ('Id1','Id2');
```

However, using the same streams, and assuming Id2 is relatively low cardinality, the following change in grouping order will significantly improve performance.

```
group q by ('Id2','Id1');
```

Id1 is now a sub-group, so the grouping algorithm is effectively working with chunks of data of lower cardinality.

Network Traffic and Latency

You might not think there's much you can do about network latency, but there are ways to reduce traffic.

Reduce Network Round Trips

Consider the number of network round trips your query might initiate. There are techniques to reduce network usage. This is especially important for mobile, where network latency can be high.

An example is faceting in a dashboard. Say you are using SAQL queries to display grouped values in a list selector, but you want the displayed values to look different (for example, you might want to show dates differently). You might choose to add an intermediate step to filter the stream based on the list selector values in order to display your prefered text. However, this adds an extra network round trip, so it's not an optimal solution.

In this case, a better solution might be to ensure your data values—those used in the list selector—are those you actually want, and have the data transformed appropriately at load time via the ELT process. See Use the ELT Process.

Redundant Filters

Is your query doing more work than it needs to? Check to see if you have redundant filters.

Optimizing Multiple Filters

Logically, it's easy to write multiple filters to achieve your goal, but often you end up with redundant filters. It's even possible to generate redundant filters when setting up binding and faceting.

```
q = load "something";
q = filter q by date('ProcDate_Day') in ["current year".."current year"];
q = filter q by date('ProcDate_Day') in ["5 years ago".."current year"];
q = group q by 'ProdDescrip';
q = foreach q generate 'ProdDescrip' as 'Prod Desc', sum('CC_cost') as 'Cost';
q = limit q 2000;
```

Even though the filters in this example occur before projection—before the foreach statement—and so are highly optimized, the second filter is redundant and so causes unnecessary work for the query engine. Why is it redundant? The results will be the same even without the "5 years ago" filter.

Note: Analytics does have a sophisticated algorithm for removing redundancy in filters, but it can't catch all cases so it's good practice to avoid redundancy.

Use the ELT Process

Is your dataset set up correctly for what you're trying to do? You may be doing unnessesary work in your queries.

The Extract, Load, and Transform Process Can Set Your Queries Up For Success

When importing your dataset via the ELT process, it's important to ensure your dataset is optimized for likely queries. The ELT process allows the creation of derived fields using calculations based on the current dataset, or even other derived fields.

If you find yourself writing queries with a case statement in the foreach projection, then it's possible your dataset could be optimised. For example, the following query changes the value JP to JAPAN in the output stream:

```
q1 = foreach q1 generate (case when 'GEO' == \"JP\" then \"Japan\" else 'GEO' end) as 'GEO;
```

You might find yourself doing this in multiple queries, which cumulatively is a performance hit. It makes better sense to have the dataset reflect the required data accurately. In your ELT process, use the *computeExpression* transformation, and add your case statement in the *saqlExpression* SAQL query. For example:

```
"action": "computeExpression",
    "parameters": {
        "source": "Opportunity_Data",
        "mergeWithSource": true,
        "computedFields": [
            {
                "name": "GEO",
                "type": "Text",
                "label": "GEO"
                "saqlExpression":
                "case
                   when 'GEO' == \"JP\" then \"Japan\"
                   else 'GEO'
                   end"}
```

```
        ]
    }
```

Now the GEO field in your dataset will contain Japan rather than JP. Your queries no longer need the CASE statement, and execute more efficiently.

See the Analytics Data Integration Guide.

Multi-Value Dimensions

If you use picklists, and find your queries are slow, consider the impact of multi-value dimensions.

Multi-Value Dimensions in Projections or Grouping

Multi-valued dimensions (for example, those used in multi-select picklists) may cause poor performance because **multi-value field behavior is undefined for group by or foreach**. Also, multi-value dimensions are not indexed, so queries that reference multi-valued dimensions will therefore require scanning of dimensions, which could slow performance. This is especially true when using multi-level grouping.

For these reasons, use of multi-value fields in anything other than filters is strongly discouraged.

Important: If you have bad performance due to multi-value fields used in foreach or group by, rewrite your query so multi-value fields are referenced only in filters.