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
Quoted String Escape Sequences ..................................... 5
Special Characters ...................................................... 5
Comments ................................................................. 6

SAQL OPERATORS .......................................................... 7
Arithmetic Operators ..................................................... 7
Comparison Operators .................................................. 7
String Operators ......................................................... 8
Logical Operators ....................................................... 8
Null Operators ........................................................... 9

SAQL STATEMENTS ......................................................... 10
load ................................................................. 10
filter ................................................................. 10
foreach .............................................................. 11
group and cogroup .................................................... 12
union ................................................................. 14
order ................................................................. 14
limit ................................................................. 15
offset ................................................................. 16

SAQL FUNCTIONS .......................................................... 17
Aggregate Functions ..................................................... 17
Date Functions ......................................................... 19
coalesce() ............................................................. 22
Use SAQL (Salesforce Analytics Query Language) to access data in Analytics Cloud datasets. Analytics Cloud uses SAQL behind the scenes in lenses, dashboards, and explorer to gather data for visualizations. Users don’t write SAQL statements, they use the UI and the app makes the queries.

Developers can write SAQL to directly access Analytics Cloud data via:

- Wave REST API
  Build your own app to access and analyze Analytics Cloud 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
- Analytics Cloud Dashboard JSON Reference
If you’re using Google Chrome to work with SAQL and Salesforce Analytics Cloud, 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
tip
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:

```plaintext
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 bold throws an error:

```plaintext
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:

```plaintext
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.
Quoted String Escape Sequences

Strings can be escaped with the backslash character. You can use the following string escape sequences:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>'</td>
<td>One single-quote character</td>
</tr>
<tr>
<td>&quot;</td>
<td>One double-quote character</td>
</tr>
<tr>
<td>\</td>
<td>One backslash character</td>
</tr>
</tbody>
</table>

Special Characters

Certain characters have special meanings in SAQL.

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Semicolon</td>
<td>Used to terminate statements.</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
<td>Used to quote identifiers.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
<td>Used to quote strings.</td>
</tr>
<tr>
<td>()</td>
<td>Parentheses</td>
<td>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.</td>
</tr>
<tr>
<td>[]</td>
<td>Brackets</td>
<td>Used to denote arrays. For example, this is an array of strings: <code>[ &quot;this&quot;, &quot;is&quot;, &quot;a&quot;, &quot;string&quot;, &quot;array&quot; ]</code> Also used for referencing a particular member of an object. For example, <code>em[ 'miles' ]</code>, which is the same as <code>em.miles</code>.</td>
</tr>
<tr>
<td>.</td>
<td>Period</td>
<td>Used for referencing a particular member of an object. For example, <code>em.miles</code>, which is the same as <code>em[ 'miles' ]</code>.</td>
</tr>
<tr>
<td>::</td>
<td>Two colons</td>
<td>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: <code>data = foreach data generate left::airline as airline</code></td>
</tr>
<tr>
<td>..</td>
<td>Two periods</td>
<td>Used to separate a range of values. For example:</td>
</tr>
<tr>
<td>Character</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( c = \text{filter} \ b \text{ by &quot;the_date&quot; in [&quot;2011-01-01&quot;..&quot;2011-01-31&quot;]}; )</td>
</tr>
</tbody>
</table>

### Comments

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

You can put a comment on its own line:

```sql
--Load a data stream.
\[load\] "myData";
```

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

```sql
\[load\] "myData"; --Load a data stream.
```

You can comment out a SAQL statement:

```sql
--The following line is commented out:
--\[load\] "myData";
```
## Arithmetic Operators

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>-</td>
<td>Minus</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
</tr>
</tbody>
</table>

## 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equals</td>
<td>True if the operands are equal. String comparisons that use the equals operator are case-sensitive.</td>
</tr>
<tr>
<td>!=</td>
<td>Not equals</td>
<td>True if the operands aren’t equal.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>True if the left operand is less than the right operand.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or equal</td>
<td>True if the left operand is less than or equal to the right operand.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>True if the left operand is greater than the right operand.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or equal</td>
<td>True if the left operand is greater than or equal to the right operand.</td>
</tr>
<tr>
<td>matches</td>
<td>Matches</td>
<td>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 &quot;LA&quot;;</td>
</tr>
</tbody>
</table>
### SAQL Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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:  

```sql
al = 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:  

```sql
al = 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 (+).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Concatenate</td>
</tr>
</tbody>
</table>

**Example:** To combine the year, month, and day into a value that’s called `CreatedAt`:

```sql
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.
### SAQL Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code> (and)</td>
<td>Logical AND</td>
<td>See table.</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>` (or)</td>
</tr>
<tr>
<td><code>!</code> (not)</td>
<td>Logical NOT</td>
<td>See table.</td>
</tr>
</tbody>
</table>

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

#### Logical Operators

|   |   | 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 |

#### Null Operators

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

### Null Operators

Use null operators to test whether a value is null.

Null operators can return true or false.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is null</code></td>
<td>Is null</td>
<td>True when the value is null.</td>
</tr>
<tr>
<td><code>is not null</code></td>
<td>Is not null</td>
<td>True when the value is not null.</td>
</tr>
</tbody>
</table>
**SAQL STATEMENTS**

### load

**Syntax**

```
result = load dataset;
```

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

If you’re working in Dashboard JSON, `dataset` can be either the containerId/versionId or the dataset name from the UI. It’s a good idea to use the dataset name (also called an alias) because the app substitutes it with the correct version of the dataset.

If you’re working in Wave 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":
  ```saql
  b = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
  ```

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

### filter

**Syntax**

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

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

**Usage**

A predicate is a Boolean expression that uses comparison 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.

- **Example:** The following example returns only rows where the origin is ORD, LAX, or LGA:
  ```saql
  al = 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:
\[y = \text{filter } x \text{ by dest == "LAX" || miles > 1500;}\]

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:
```sql
a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
svm 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
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, such as the fields that were used as the grouping keys. 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;`

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:

```sql
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 sumlval;
```

SEE ALSO:
- **Statements**
- **Aggregate Functions**

### 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**

```sql
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:**

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

or

```sql
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:

\[
\text{result} = \text{cogroup} \ \text{rows} \ \text{by} \ \text{expression}[, \ \text{rows} \ \text{by} \ \text{expression} \ ...];
\]

This example is a simple cogroup operation on 2 datasets:

\[
a = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = \text{load} \ "0Fbyy000000002qCAA/0Fcyy000000002WCAQ";
a = \text{cogroup} \ a \ \text{by} \ \text{carrier}, \ b \ \text{by} \ \text{carrier};
\]

You can cogroup more than 2 datasets:

\[
\text{result} = \text{cogroup} \ a \ \text{by} \ \text{keya}, \ b \ \text{by} \ \text{keyb}, \ c \ \text{by} \ \text{keyc};
\]

This example performs a cogroup operation:

\[
z = \text{cogroup} \ x \ \text{by} \ \{\text{day,origin}\}, \ y \ \text{by} \ \{\text{day,airport}\};
\]

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:

\[
a = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = \text{cogroup} \ a \ \text{by} \ \text{ClosedDate}, \ b \ \text{by} \ \text{CreatedDate};
c = \text{foreach} \ b \ \text{generate} \ \text{sum}(a.\text{Amount}) \ \text{as} \ \text{Amount};
\]

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

\[
a = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
a = \text{filter} \ a \ \text{by} \ \"\text{region}\" \ \text{in} \ \{\"West\\};
a = \text{filter} \ a \ \text{by} \ \"\text{status}\" \ \text{in} \ \{\"closed\\};
b = \text{filter} \ a \ \text{by} \ \"\text{year}\" \ \text{in} \ \{2014\};
c = \text{filter} \ a \ \text{by} \ \"\text{year}\" \ \text{in} \ \{2015\};
d = \text{cogroup} \ b \ \text{by} \ \{\"state\\}, \ c \ \text{by} \ \{\"state\\};
d = \text{foreach} \ d \ \text{generate} \ \text{"state" as} \ \text{"state"}, \ \text{sum}(b.\text{Amount}) \ \text{as} \ \"\text{Amount}_2014\", \ \text{sum}(c.\text{Amount}) \ \text{as} \ \"\text{Amount}_2015\";
\]

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

\[
a = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = \text{cogroup} \ a \ \text{by} \ \text{ClosedDate}, \ a \ \text{by} \ \text{CreatedDate};
c = \text{foreach} \ b \ \text{generate} \ \text{sum}(a.\text{Amount}) \ \text{as} \ \text{Amount};
\]

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:

\[
\text{sum}(\text{inputSide["myMeasure"]})
\]
\[
\text{sum}(\text{inputSide::myMeasure})
\]
\[
\text{sum}(\text{inputSide.myMeasure})
\]

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

\[
a = \text{load} \ "0Fbxx000000002qCAA/0Fcxx000000002WCAQ";
b = \text{load} \ "0Fbyy000000002qCAA/0Fcyy000000002WCAQ";
\]
c = cogroup a by x, b by y;
d = foreach c generate a.x as x, a.y as y, \text{sum}(a.miles) as miles;

This query isn’t valid because \text{miles} doesn’t specify which side it is coming from:

\begin{verbatim}
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;
\end{verbatim}

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

\begin{verbatim}
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;
\end{verbatim}

**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:

\begin{verbatim}
result \texttt{=} cogroup \texttt{rows by expression [left | right | full], rows by expression;}
\end{verbatim}

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

Example: 
\begin{verbatim}
z = cogroup x by (day,origin) left, y by (day,airport);
\end{verbatim}

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

\begin{verbatim}
result \texttt{=} cogroup a by keya left, b by keyb right, c by keyc;
\end{verbatim}

\textit{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

\begin{verbatim}
result \texttt{=} union resultSetA, resultSetB [, resultSetC ...];
\end{verbatim}

**order**

Sorts by one or more attributes.
Syntax

```
result = order rows by attribute [ asc | desc ];
result = order rows by (attribute [ asc | desc ], attribute [ 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 attributes, use this syntax:

  ```
a = load "0Fbxx00000000002qCAA/0Fcxx00000000002WCAQ";
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’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 "0Fbxx00000000002qCAA/0Fcxx00000000002WCAQ";
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 "0Fbxx00000000002qCAA/0Fcxx00000000002WCAQ";
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:

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

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

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

SEE ALSO:

Statements
order

offset

Paginates values from query results.

Syntax

```sql
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:

```sql
a = load "0Fbxx00000002qCAA/0Fcxx00000002WCAQ";
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
## 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.

This table lists the aggregate functions that are supported:

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>avg()</code> or <code>average()</code></td>
<td>Returns the average value of a numeric field. For example, to calculate the average number of miles:</td>
</tr>
<tr>
<td></td>
<td><code>a1 = group a by (origin, dest);</code></td>
</tr>
<tr>
<td></td>
<td><code>a2 = foreach a1 generate origin as origin,</code></td>
</tr>
<tr>
<td></td>
<td><code>dest as destination, average(miles) as miles;</code></td>
</tr>
<tr>
<td><code>count()</code></td>
<td>Returns the number of rows that match the query criteria. For example, to calculate the number of carriers:</td>
</tr>
<tr>
<td></td>
<td><code>q = foreach q generate 'carrier' as 'carrier', count() as 'count';</code></td>
</tr>
<tr>
<td></td>
<td>The <code>count()</code> function operates on streams that were inputs to the <code>group</code> or <code>cogroup</code> statements. It doesn’t operate on the newly grouped stream or on an ungrouped stream.</td>
</tr>
<tr>
<td></td>
<td><code>a = load &quot;0Fcyy000000002qCAA/0Fcyy000000002WCAQ&quot;;</code></td>
</tr>
<tr>
<td></td>
<td><code>a1 = group a by (Year);</code></td>
</tr>
<tr>
<td></td>
<td><code>q = foreach a1 generate count(a) as countYear, count() as count, Year as year;</code></td>
</tr>
<tr>
<td></td>
<td><code>q = limit q 20;</code></td>
</tr>
<tr>
<td></td>
<td>You can’t pass <code>a1</code> to the <code>count()</code> function because it’s a newly grouped stream.</td>
</tr>
<tr>
<td>Aggregate Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>first()</strong></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td><code>a1 = group a by (origin, dest);</code></td>
</tr>
<tr>
<td></td>
<td><code>a2 = foreach a1 generate origin as origin, dest as destination, first(miles) as miles;</code></td>
</tr>
<tr>
<td><strong>last()</strong></td>
<td>Returns the value for the last tuple. For example, to compute the distance between each combination of origin and destination:</td>
</tr>
<tr>
<td></td>
<td><code>a1 = group a by (origin, dest);</code></td>
</tr>
<tr>
<td></td>
<td><code>a2 = foreach a1 generate origin as origin, dest as destination, last(miles) as miles;</code></td>
</tr>
<tr>
<td><strong>max()</strong></td>
<td>Returns the maximum value of a field. This function takes only a measure as an argument. It can't take a dimension.</td>
</tr>
<tr>
<td><strong>min()</strong></td>
<td>Returns the minimum value of a field. This function takes only a measure as an argument. It can't take a dimension.</td>
</tr>
<tr>
<td><strong>sum()</strong></td>
<td>Returns the sum of a numeric field.</td>
</tr>
<tr>
<td></td>
<td><code>a = load &quot;0Fbxx000000002qCAA/0Fcxx000000002WCAQ&quot;;</code></td>
</tr>
<tr>
<td></td>
<td><code>a = filter a by dest in [&quot;ORD&quot;, &quot;LAX&quot;, &quot;ATL&quot;, &quot;DFW&quot;, &quot;PHX&quot;, &quot;DEN&quot;, &quot;LGA&quot;];</code></td>
</tr>
<tr>
<td></td>
<td><code>a = group a by carrier;</code></td>
</tr>
<tr>
<td></td>
<td><code>b = foreach a generate carrier as airline, sum(miles) as miles;</code></td>
</tr>
<tr>
<td><strong>unique()</strong></td>
<td>Returns the count of unique values. For example, to find how many origins and destinations a carrier flies from:</td>
</tr>
<tr>
<td></td>
<td><code>a1 = group a by carrier;</code></td>
</tr>
<tr>
<td></td>
<td><code>a2 = foreach a1 generate carrier as carrier, unique(origin) as origins, unique(dest) as destinations;</code></td>
</tr>
</tbody>
</table>
To specify dates in a SAQL query, use date functions and relative date keywords.

## Functions

This table lists SAQL date functions:

<table>
<thead>
<tr>
<th>Date Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date(year, month, day)</code></td>
<td>Returns a date. Specify three dimensions of a date in the following order: year, month, day. For example:</td>
</tr>
<tr>
<td></td>
<td><code>date('OrderDate_Year', 'OrderDate_Month', 'OrderDate_Day')</code></td>
</tr>
<tr>
<td><code>dateRange(startArray_y_m_d, endArray_y_m_d)</code></td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td><code>dateRange([1970, 1, 1], [1970, 1, 31])</code></td>
</tr>
<tr>
<td><code>daysBetween(date1, date2)</code></td>
<td>Returns the number of days between two dates as an integer. The <code>daysBetween()</code> function can’t take dimensions as arguments directly. Pass <code>toDate()</code> and <code>now()</code> functions as arguments.</td>
</tr>
<tr>
<td></td>
<td><code>q = foreach q generate daysBetween(toDate(OrderDate, &quot;yyyy-MM-dd&quot;), now()) as daysToShip;</code></td>
</tr>
<tr>
<td></td>
<td><code>q = foreach q generate daysBetween(toDate(OrderDate, &quot;yyyy-MM-dd&quot;), toDate(ShipDate, &quot;yyyy-MM-dd&quot;)) as daysToShip;</code></td>
</tr>
<tr>
<td></td>
<td><code>q = foreach q generate daysBetween(toDate(OrderDate_Year + &quot;:&quot; + OrderDate_Month + &quot;:&quot; + OrderDate_Day, &quot;yyyy:MM:dd&quot;), toDate(ShipDate_Year + &quot;:&quot; + ShipDate_Month + &quot;:&quot; + ShipDate_Day, &quot;yyyy:MM:dd&quot;)) as daysToShip;</code></td>
</tr>
<tr>
<td><code>now()</code></td>
<td>Returns current datetime in UTC (GMT). This function is valid in a <code>foreach</code> statement only.</td>
</tr>
<tr>
<td></td>
<td><code>q = foreach q generate now() as now;</code></td>
</tr>
<tr>
<td></td>
<td>This function is commonly used in <code>daysBetween()</code> and <code>toString()</code> functions.</td>
</tr>
</tbody>
</table>
**Date Functions**

<table>
<thead>
<tr>
<th>Date Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>toDate(string [, formatString])</code></td>
<td>Converts a string to a date. If a <code>formatString</code> argument isn't provided, the function uses the format <code>yyyy-MM-dd HH:mm:ss</code>.</td>
</tr>
<tr>
<td>q = foreach q generate toDate(OrderDate);</td>
<td></td>
</tr>
<tr>
<td>q = foreach q generate toDate(OrderDate_Day + &quot;-&quot; + OrderDate_Month + &quot;-&quot; + OrderDate_Year, dd-MM-yyyy );</td>
<td></td>
</tr>
</tbody>
</table>
| 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_sec)` returns '1970-01-01 00:00:00'. |

| 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; |

**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
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).

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

**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:

```sql
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**:

```sql
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**:

```sql
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.

```sql
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 `yyyyMMdd`.

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

### coalesce()

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

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

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

```sql
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.

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