Analytics SAQL Developer Guide

Salesforce, Winter '23

'23

@salesforcedocs

Last updated: November 3, 2022

CONTENTS

OVERVIEW

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

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

- CRM Analytics REST API

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

- Dashboard JSON

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

- Compare Table

  Use SAQL to perform calculations on data in your tables and add the results to a new column.

- Transformations During Data Flow

  Use SAQL to perform manipulations or calculations on data when bringing it in to CRM Analytics.

SEE ALSO:

Analytics REST API Developer Guide

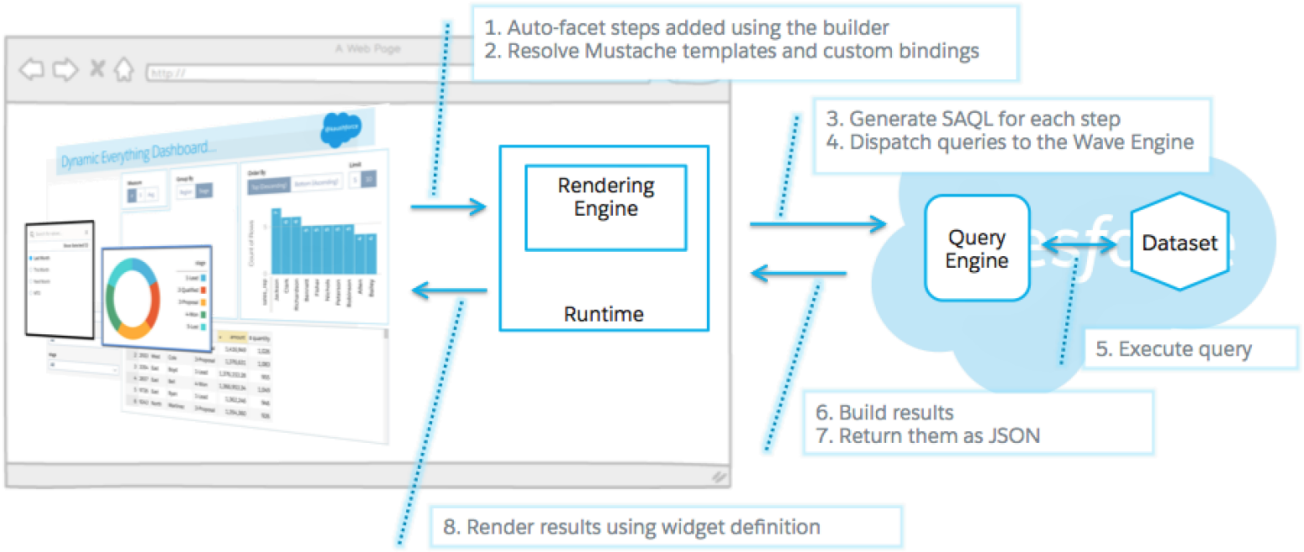
Analytics Dashboard JSON Developer Guide

Introduction

Most actions you take in Analytics result in one or more SAQL queries. Every lens, dashboard, and explorer action generates and executes a SAQL query to build the data needed for the visualization.

Analytics evaluates queries, widgets, and layouts to render a dashboard. Behind every widget is a SAQL query which is sent the query engine for execution. The resulting data is passed to the charting library, which renders it using corresponding widget definitions. SAQL is influenced by the Apache Pig Latin (pigql) syntax, but their implementations differ, and they are not compatible.

How the components fit together



Developers can write SAQL to access Analytics data, either via the Analytics REST API, or by creating and 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 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.

Use SAQL in the CRM Analytics Dashboard

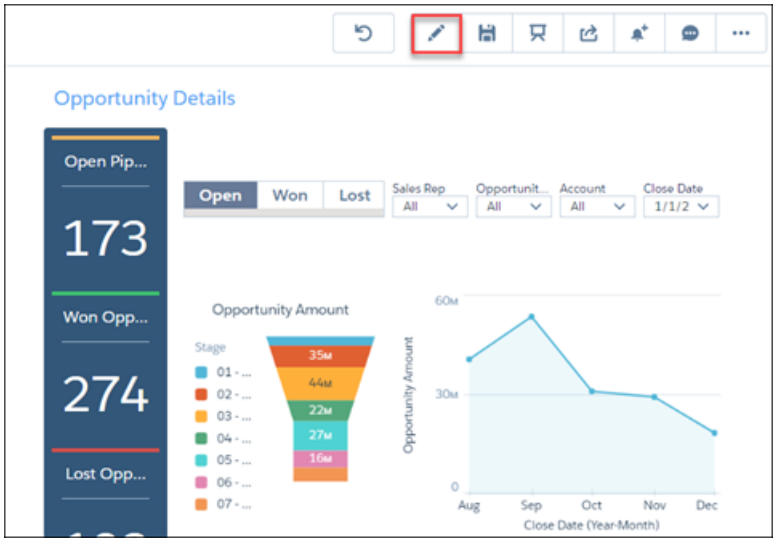
Use the CRM Analytics Studio user interface to modify existing SAQL queries or write new ones. Writing SAQL queries in the user interface is the easiest way to get started.

Every component in CRM Analytics uses SAQL behind the scenes. You can build a widget in a dashboard, then switch to the SAQL view to see the SAQL query for the widget. Or, you can create a lens while exploring a dataset, then switch to the SAQL view to see the SAQL query for the lens.

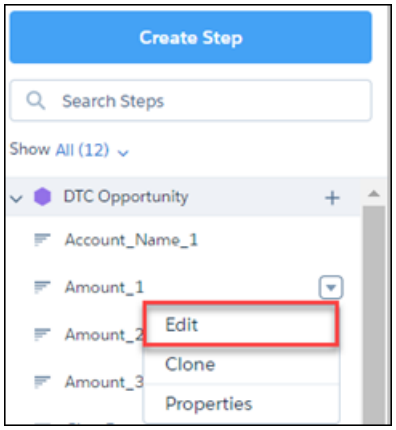
Let's look at the query generated by a widget in a dashboard.

Note: After you edit the SAQL for a widget, you may not be able to go back to the dashboard view, depending on how complex the SAQL query is.

1. In your Salesforce org, open CRM Analytics Studio, then open a dashboard. For example, open Opportunity Details.
2. Click **Edit**.



3. Click a query to edit, for example Amount_1, then click **Edit** in the dropdown list.



4. Click **SAQL Mode** to display the SAQL query.

5. View the SAQL query.

   Here is the SAQL query for our example:

```
q = load "DTC_Opportunity_SAMPLE";
q = filter q by 'Closed' == "false";
q = group q by all;
```

```
q = foreach q generate sum('Amount') as 'sum_Amount';
q = limit q 2000;
```

6. Edit the query, then click **Run Query** to run the new query. For example, you could change the `sum` to `average`.

Enable SAQL Logs in the Browser

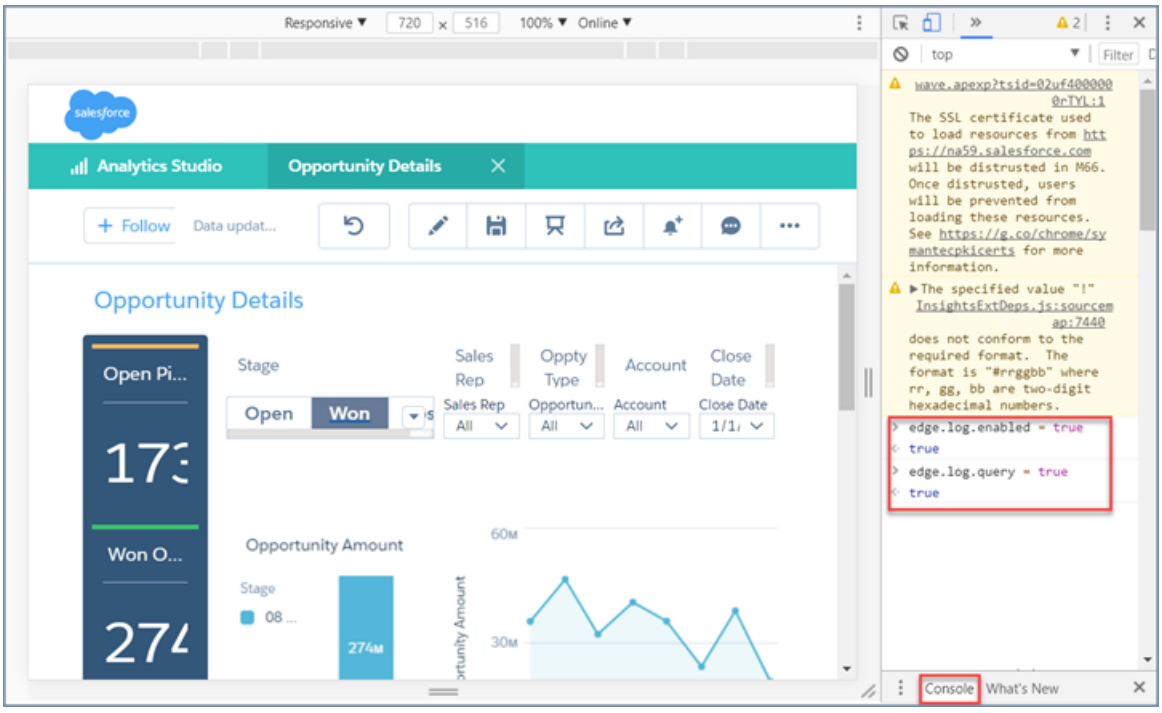
If you're using Google Chrome to work with SAQL and Einstein CRM Analytics, you can turn on SAQL logs.

Note: SAQL Logs in the browser are no longer supported. To see how your SAQL queries run in the dashboard, use the Dashboard Inspector. You can also right-click the dashboard in the browser and select **Inspect**.

Turning on SAQL logs in the browser prints queries in the Developer Tools Console. This lets you see what SAQL is generated by CRM Analytics dashboards and lenses. This action doesn't change server-side logs.

1. In Google Chrome, open a CRM Analytics dashboard.
2. In Google Chrome, open Developer Tools.
3. In Developer Tools, select Console.
4. In the Einstein Analtyics dashboard, elect the explore (wave.apexp) frame.
5. In the developer tools console, enter *`edge.log.enabled = true`*
6. In the developer tools console, enter *`edge.log.query = true`*

SAQL logs are enabled. The logs are displayed when a query is sent from the dashboard or lens, for example when you drill into a chart.



QUICK START

Get up to speed quickly with these easy SAQL examples.

Write Your First Query

Let's walk through each part of a simple SAQL query.

We'll create a new dashboard in a CRM Analytics org. Then we'll add a simple chart and look at the resulting SAQL.

Note: These instructions assume you are using the sample Salesforce Developer org, which includes sample datasets. If you are using a different org, you can still follow the same general instructions with your own dataset.

1. In your CRM Analytics org, create a new dashboard:
   a. Click **Create**.
   b. Click **Dashboard**.
2. In the window Choose a dashboard template, click **Blank Dashboard**, then click **Continue**.
3. Drag a chart widget to the dashboard canvas.
4. In the chart widget, click **Chart**, then select **DTC Opportunity** dataset.
5. Click the **SAQL Mode** button to launch the SAQL editor.

The SAQL editor displays the SAQL query used to fetch the data and render the chart:

```
1   q = load "DTC_Opportunity_SAMPLE";
2   q = group q by all;
3   q = foreach q generate count() as 'count';
4   q = limit q 2000;
```

Let's take a look at each line in the query.

Line Number	Description
1	`q = load "DTC_Opportunity_SAMPLE";` This loads the dataset that you chose when you created the chart widget. You can use the variable `q` to access the dataset in the rest of your SAQL statements.

Line Number	Description
2	`q = group q by all;` In some queries, you want to group by a certain field, for example Account ID. In our case we didn't specify a grouping when we created the chart. Use `group by all` when you don't want to group data.
3	`q = foreach q generate count() as 'count';` This generates the output for our query. In this simple example, we just count the number of lines in the DTC Opportunity dataset.
4	`q = limit q 2000` This limits the number of results that are returned to 2000. Limiting the number of results can improve performance. However if you want `q` to contain more than 2000 results, you can increase this number.

You can click **Back** to go back to the chart. You can use the UI to make modifications to the chart, then view the resulting SAQL.

Create a Derived Measure

Perform calculations on existing measures and use the result to create a new, or derived, measure.

CRM Analytics calculates the value of derived measures at run time using the values from other fields.

Note: You can also create a derived measure in a dataflow rather than at runtime using SAQL. Measures created during a dataflow are calculated when the data is imported and may result in better performance.

Example - Calculate the Time to Win

Suppose that you have an Opportunities dataset with the Close Date and Open Date fields. You want to see the number of days it took to win the opportunity. Use Close_Date_day_epoch and Created_Date_day_epoch to create a derived measure called Time to Win: `('Close_Date_day_epoch'- 'Created_Date_day_epoch') as 'Time to Win'`.

The field Time to Win is calculated at run time:

```
q = load "Opportunities";
q = foreach q generate 'Close_Date_day_epoch' as 'Close_Date_day_epoch',
'Created_Date_day_epoch' as 'Created_Date_day_epoch', 'Opportunity_Name' as
'Opportunity_Name', ('Close_Date_day_epoch'- 'Created_Date_day_epoch') as 'Time to Win';
```

The resulting table contains the number of days to win each opportunity:

Close Date (Epoch days)	Created Date (Epoch days)	Opportunity Name	Time to Win
16,762	16,707	Opportunity for Wood9	55
16,886	16,750	Opportunity for Jefferson17	136
17,066	16,942	Opportunity for McLaughlin130	124

Create a Derived Dimension

Perform string manipulations on existing dimensions to create a new, or derived, dimension.

CRM Analytics creates derived dimensions at run time.

Note: You can also create a derived dimension in a dataflow rather than at runtime.

Example - Create a Field with City and State

Suppose that you have an Opportunities dataset with a City and a State field. You want to create a single field containing both city and state. Use SAQL to create a derived dimension.

```
q = load "Ops";
q = foreach q generate 'Account' as 'Account', 'Amount' as 'Amount', 'City' + "-" + 'State'
 as 'City - State';
```

The resulting table contains city and state in the same field.

Account	Amount	City - State
Shoes2Go	1.5	Springfield-Illinois
FreshMeals	2	Springfield-Alabama
ZipBikeShare	1.1	Springfield-Missouri
Shoes2Go	3	Springfield-Georgia

EXAMPLES

These hands-on SAQL examples walk you through writing a query to retrieve data

Analyze Your Data Over Time

Use SAQL date functions for advanced time-based analysis.

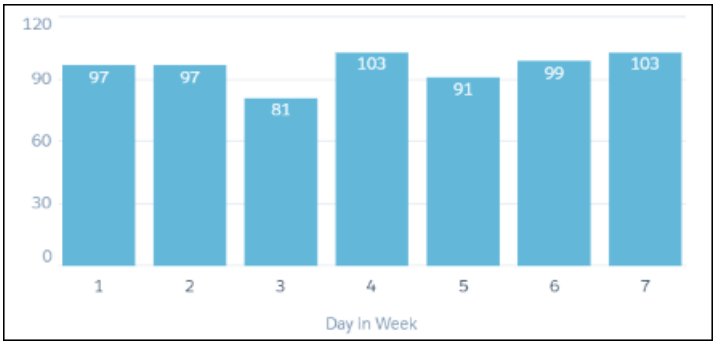
Note: You can use date filters in the dashboard for basic time-based analysis, for example to calculate month-to-date amounts. You can also use window functions in the dashboard for basic date range calculations, such as calculating the change in year-over-year earnings

Example - on Which Weekday Do Customers Send the Most Emails?

Suppose that you want to see which day of the week your customers are most active on email. This information allows you to better target your email campaigns. Use `day_in_week()` on the Mail_sent_sec_epoch field to calculate the day of the week, then count the number of records for each day.

```
q = load "DTC_Opportunity_SAMPLE";
q = foreach q generate day_in_week(toDate(Mail_sent_sec_epoch)) as 'Day in Week';
q = group q by 'Day in Week';
q = foreach q generate 'Day in Week', count() as 'count';
```

In this case, email traffic is slightly higher on day 4 (Wednesday) and day 7 (Sunday).



SEE ALSO:

Date Functions

Calculate How Long Activities Take

Use `daysBetween()` and `date_diff()` to calculate the difference between two dates or times.

Example: Display the Number of Days Since an Opportunity Opened

Suppose that you have an opportunity dataset with the account name and the epoch seconds fields:

Account	OrderDate_sec_epoch
Shoes2Go	1,521,504,003
FreshMeals	1,521,158,403
ZipBikeShare	1,518,739,203

You want to see how many days ago an opportunity was opened. Use `daysBetween()` and `now()`. Use `toDate()` to convert the order date epoch seconds to a date format that can be passed to `daysBetween()`.

```
q = load "OpsDates1";

q = foreach q generate Account, daysBetween(toDate(OrderDate_sec_epoch), now()) as
'daysOpened';
```

The resulting data stream displays the number of days since the opportunity was opened.

Account	daysOpened
Shoes2Go	66
FreshMeals	70
ZipBikeShare	98

Example - How Many Weeks Did Each Opportunity Take to Close?

Use `date_diff()` with *`datepart`* = `week` to calculate how long, in weeks, it took to close each opportunity.

```
q = load "DTC_Opportunity";
q = foreach q generate date_diff("week", toDate(Created_Date_sec_epoch),
toDate(Close_Date_sec_epoch) ) as 'Weeks to Close';
q = order q by 'Weeks to Close';
```

SEE ALSO:

daysBetween()

date_diff()

Display the Opportunities Closed This Month

Use relative date ranges to filter opportunities closed in the current month.

Example: Display Opportunities Closed This Month

Suppose that you want to see which opportunities closed this month. Your data includes the account name, the close date fields, and the epoch seconds field.

Account	CloseDate (Year)	CloseDate (Month)	CloseDate_sec_epoch	CloseDate (Day)
Shoes2Go	2018	05	1,526,774,403	20
FreshMeals	2018	03	1,522,368,003	30
ZipBikeShare	2018	02	1,519,516,803	25

Use `date()` to generate the close date in date format. Then use relative date ranges to filter opportunities closed in the current month.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
month" .. "current month"];
q = foreach q generate Account;
```

If the query is run in May 2018, the resulting data stream contains one entry:

Account
Shoes2Go

To add the close date in a readable format, use `toDate()`.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
month" .. "current month"];
q = foreach q generate Account, toDate('CloseDate_sec_epoch') as 'Close Date';
```

The resulting data stream includes the full date and time of the close date.

Account	Close Date
Shoes2Go	2018-05-20 00:00:03

You can also display just the month and day of the close date.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
month" .. "current month"];
q = foreach q generate Account, 'CloseDate_Month' + "/" + 'CloseDate_Day' as 'Close Date';
```

The resulting data stream contains the month and day of the close date.

Account	Close Date
Shoes2Go	05/20

SEE ALSO:

Time-Based Filtering

Forecast Future Data Points with `timeseries`

Use existing data to predict what might happen in the future.

Example - How Many Tourists Will Visit Next Year?

Suppose that you run a chain of retail stores, and the number of tourists in your city affect your sales. Use `timeseries` to predict how many tourists will come to your city next year:

```
q = load "TouristData";
q = group q by ('Visit_Year', 'Visit_Month');
q = foreach q generate 'Visit_Year', 'Visit_Month', sum('NumTourist') as 'sum_NumTourist';

-- If your data is missing some dates, use fill() before using timeseries()
-- Make sure that the dateCols parameter in fill() matches the dateCols parameter in
timerseries()
q = fill q by (dateCols=('Visit_Year','Visit_Month', "Y-M"));
```

```
-- Use timeseries() to predict the number of tourists.
q = timeseries q generate 'sum_NumTourist' as Tourists with (length=12,
dateCols=('Visit_Year','Visit_Month', "Y-M"));

q = foreach q generate 'Visit_Year' + "~~~" + 'Visit_Month' as 'Visit_Year~~~Visit_Month',
 Tourists;
```

Use a timeline chart and set a predictive line to see the calculated future data. The resulting graph shows the likely number of tourists in the future.

SEE ALSO:

timeseries

Combine Data from Multiple Data Streams with `cogroup`

You can combine data from two or more data streams into a single data stream using `cogroup`. The data streams must have at least one common field.

Example - Inner `cogroup`

Suppose that you want to understand how much time your reps spend meeting with each account. Is there a relationship between spending more time and winning an account? Are some reps spending much more or much less time than average? To answer these questions, first combine meeting data with account data using `cogroup`.

Suppose that you have a dataset of meeting information from the Salesforce Event object. In this example, your reps have had six meetings with four different companies. The Meetings dataset has a MeetingDuration column, which contains the meeting duration in hours.

#	Company	MeetingDuration
1	Shoes2Go	2
2	FreshMeals	3
3	ZipBikeShare	4
4	Shoes2Go	5
5	FreshMeals	1
6	ZenRetreats	6

The account data exists in the Salesforce Opportunity object. The Ops dataset has an Account, Won, and Amount column. The Amount column contains the dollar value of the opportunity, in millions.

#	Account	Won	Amount
1	Shoes2Go	1	1.5
2	FreshMeals	1	2
3	ZipBikeShare	1	1.1
4	Shoes2Go	0	3
5	FreshMeals	1	1.4
6	ZenRetreats	0	2

To see the effect of meeting duration on opportunities, you start by combining these two datasets into a single data stream using `cogroup`.

```
q = cogroup ops by 'Account', meetings by 'Company';
```

Internally (you cannot see these results yet), the resulting cogrouped data stream contains the following data. Note how the data streams are rolled up on one or more dimensions.

```
(1,{(Shoes2Go,2,), (Shoes2Go,5)},{(Shoes2Go,1,1.5), (Shoes2Go,0,3})

(2,{(FreshMeals,3), (FreshMeals, 5)},{(FreshMeals,1,2) (FreshMeals, 1, 1.4)})

(3,{(ZipBikeShare,4)},{(ZipBikeShare,1, 1.1)})

(4,{(ZenRetreats,6)},{(ZenRetreats,0, 2)})
```

Now the datasets are combined. To see the data, you create a projection using `foreach`:

```
ops = load "Ops";
meetings = load "Meetings";
q = cogroup ops by 'Account', meetings by 'Company';
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
sum(meetings.'MeetingDuration') as 'TimeSpent';
```

The resulting data stream contains the sum of amount and total meeting time for each company. The sum of amount is the sum of the dollar value for every opportunity for the company.

Account	Sum of Amount	TimeSpent
Company1	4.5	7
Company2	3.4	4
Company3	1.1	4
Company4	2	6

Now that you have combined the data into a single data stream, you can analyze the effects that total meeting time has on your opportunities.

SEE ALSO:

cogroup

Replace Null Values with `coalesce()`

When you use a left outer or full outer `cogroup`, unmatched data comes through as null. Use `coalesce()` to replace null values with the value of your choice.

Example: Left Outer Cogroup with `coalesce()`

A left outer cogroup combines the right data stream with the left data stream. If a record on the left stream does not have a match on the right stream, the missing right value comes through as null. To replace null values with a different value, use `coalesce()`.

For example, suppose that you have a dataset of meeting information from the Salesforce Event object, and you join it with data from the Salesforce Opportunity object. This shows amount won with the total time spent in meetings.

```
ops = load "Ops";
meetings = load "Meetings";
q = cogroup ops by 'Account' left, meetings  by 'Company' ;
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
sum(meetings.'MeetingDuration') as 'TimeSpent';
```

It looks like we had no meetings with Zen Retreats.

Account	Sum of Amount	TimeSpent
FreshMeals	3.4	4
Shoes2Go	4.5	7
ZenRetreats	2	-
ZipBikeShare	1.1	4

Let's use `coalesce()` to change that null value to a zero.

```
ops = load "Ops";
meetings = load "Meetings";
q = cogroup ops by 'Account' left, meetings  by 'Company' ;

--use coalesce() to replace null values with zero
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
coalesce(sum(meetings.'MeetingDuration'), 0) as 'TimeSpent';
```

Account	Sum of Amount	TimeSpent
FreshMeals	3.4	4
Shoes2Go	4.5	7
ZenRetreats	2	0
ZipBikeShare	1.1	4

SEE ALSO:

cogroup

Dynamically Display Your Top Five Reps with Windowing

Windowing functions perform calculations over a dynamic range.

Example - Dynamically Display Your Top Five Reps

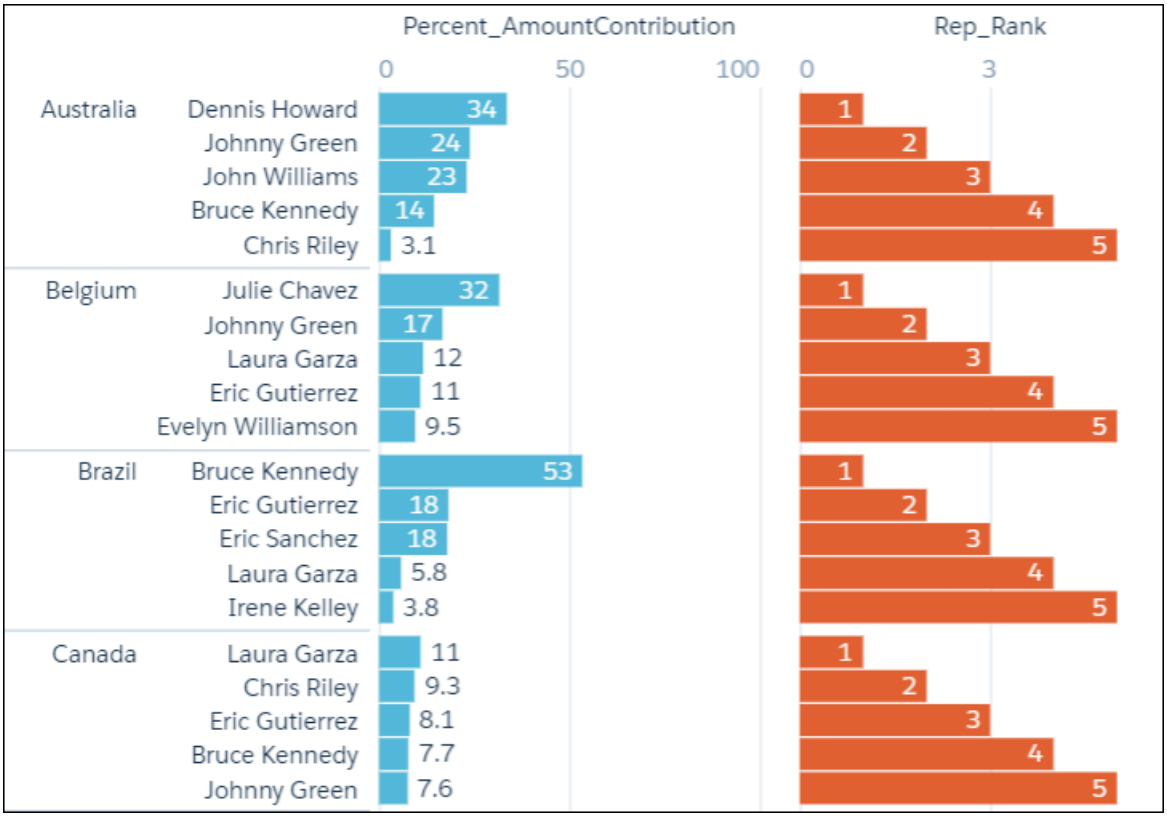
Use windowing to create a chart that dynamically displays your top-five reps for each country. The chart updates continuously as opportunities are won. The example uses windowing to calculate:

- Percentage contribution that each rep made to the total amount, partitioned by country
- Ranking of the rep's contribution, partitioned by country

These calculations let us display the top-five reps in each country.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by ('Billing_Country', 'Account_Owner');

q = foreach q generate 'Billing_Country', 'Account_Owner',

-- sum(Amount) is the total amount for a single rep in the current country
-- sum(sum('Amount') is the total amount for ALL reps in the current country
-- sum(Amount) / sum(sum('Amount') calculates the percentage that each rep contributed
-- to the total amount in the current country
((sum('Amount')/sum(sum('Amount'))


-- [..] means "include all records in the partition"
-- "by Billing_Country" means partition, or group, by country
over ([..] partition by 'Billing_Country')) * 100) as 'Percent_AmountContribution',

-- rank the percent contribution and partition by the country
rank() over ([..] partition by ('Billing_Country') order by sum('Amount') desc ) as
'Rep_Rank';

-- filter to include only the top 5 reps
q = filter q by 'Rep_Rank' <=5;
```

The resulting graph shows the top-five reps in each country and displays each rep's ranking.



Append Datasets using `union`

You can append data from two or more data streams into a single data stream using `union`. The data streams must have the same field names and structure.

To use `union`, first load the dataset and then use `foreach` to do the projection. Repeat the process with another dataset. If the two resulting data streams have an identical structure, you can append them using `union`.

Let's say that you have two opportunity datasets from different regions that you brought together using the Salesforce mulit-org connector. You want to add these datasets together to look at your pipeline as a whole.

The OppsRegion1 data stream contains these fields.

#	Account Owner	Account Type	Amount
1	Laura Palmer	Customer	8,577,295
2	Laura Garza	Customer	5,839,810
3	Dennis Howard	Customer	5,423,800
4	Nicolas Weaver	Customer	5,335,150

The OppsRegion2 data stream contains these fields.

#	Account Owner	Account Type	Amount
1	Bruce Kennedy	Partner	14,260
2	Laura Garza	Customer	18,178
3	Julie Chavez	Customer	20,493

Use `union` to combine the two data streams.

```
ops1 = load "OppsRegion1";

ops1 = foreach ops1 generate 'Account_Owner', 'Account_Type', 'Amount';

ops2 = load  "OppsRegion2";
ops2 = foreach ops2 generate 'Account_Owner', 'Account_Type', 'Amount';

-- ops1 and ops2 have the same structure, so we can use union
opps_total = union ops1, ops2;
```

The resulting data stream contains both sets of data.

#	Account Owner	Account Type	Amount
1	Laura Palmer	Customer	8,577,295
2	Laura Garza	Customer	5,839,810
3	Dennis Howard	Customer	5,423,800
4	Nicolas Weaver	Customer	5,335,150
5	Bruce Kennedy	Partner	14,260
6	Laura Garza	Customer	18,178
7	Julie Chavez	Customer	20,493

SEE ALSO:

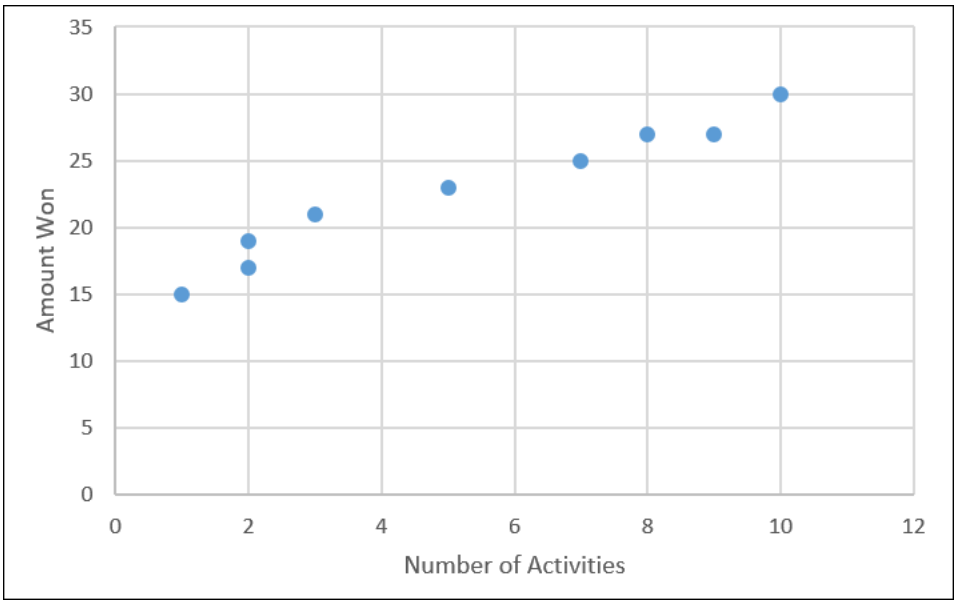
union

Calculate the Slope of the Regression Line

Use SAQL to perform linear analysis on your data to find the line that best fits the data. Then use `.regr_slope` to return the slope of this line.

Example - Calculate the Relationship Between Number of Activities and Deal Amount

Suppose that you have a dataset that includes the number of activities (such as meetings) and the won opportunity amount.



How much bigger with the deal size be for each extra activity? `regr_slope` performs a linear analysis on your data then calculates the slope (that is, the increased amount you win for each extra activity).

```
q = load "data/sales";
q = group q by all;

--trunc() truncates the result to two decimal places
q = foreach q generate trunc(regr_slope('Amount', 'NumActivities'),2) as 'Gain per Activity';
```

Based on your existing data, every extra activity that you have tends to increase the deal size by $1.45 million, on average.

Gain per Activity	1.45

SEE ALSO:

regr_slope()

Show the Top and Bottom Quartile

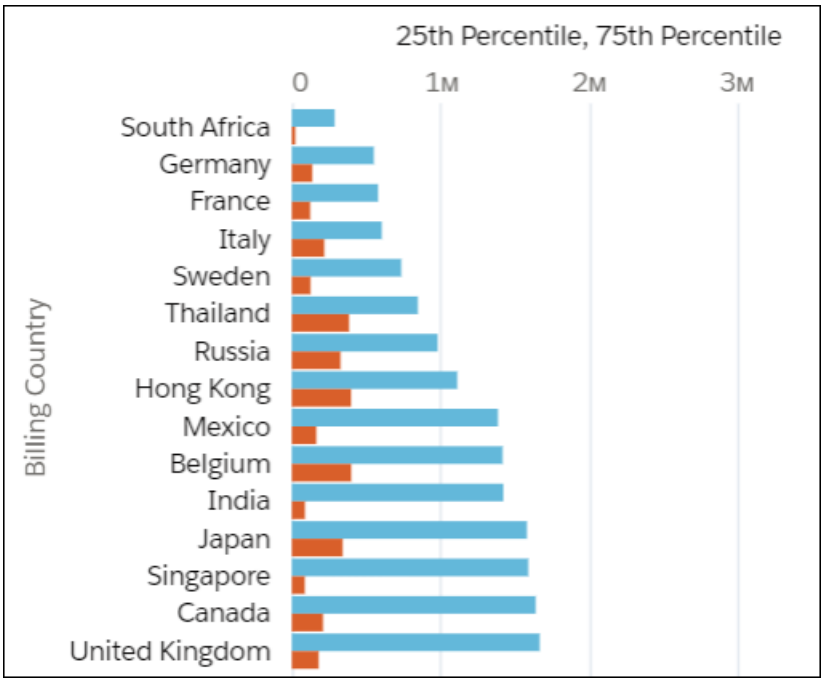
Use SAQL to calculate percentiles, like the top and bottom quartile of your data.

Example - Show Top Quartile and Bottom Quartile Deal Size by Country

Suppose that you want to see the top and bottom quartile deal size, by country. You want to see the size of the actual deal, not the interpolated (or 'average') deal size. Use `percentile_disc(.25)` and `percentile_disc(.75)`.

```
q = load "Data";
q = group q by 'Billing_Country';
q = foreach q generate 'Billing_Country' as 'Billing_Country', percentile_disc(0.25) within
 group (order by 'Amount' desc) as '25th Percentile', percentile_disc(0.75) within group
(order by 'Amount' desc) as '75th Percentile';
q = order q by '25th Percentile' asc;
```

Use a bar chart and select **Axis Mode** > **Single Axis** to show the top and bottom quartiles together.



SEE ALSO:

percentile_disc()

Calculate Grand Totals and Subtotals with the `rollup` Modifier and `grouping()` Function

Calculate subtotals of grouped data in your SAQL query using the `rollup` modifier on the `group by` statement, then work with subtotaled data using `grouping()`. For example, to see the subtotaled value of opportunities by type and lead source, roll up the type and lead source groups. Then, label the subtotals with the grouping function.

Invoking `rollup` adds rows to your query results with null values for dimensions and subtotaled results for measures. Invoking `grouping()` returns 1 if null dimension values are due to higher-level aggregates (which usually means the row is a subtotal), otherwise it returns 0.

Using `grouping()` alongside `rollup` lets you work with subtotaled data. After subtotaling data, common next steps include logically evaluating subtotaled data with a case statement. Or filtering on subtotaled data with a filter statement.

Suppose that you have an opportunity dataset, and want to see the value of deals by lead source and type. Plus, you want to see the total value of all lead sources and all types. Write a query that returns the sum of opportunity amount grouped by type and lead source. To see the value of all lead sources and all types, use `rollup` to subtotal opportunities, then use `grouping()` to label the subtotaled rows.

Example: `rollup`

Open the SAQL editor in the dashboard. Instead of grouping data by a field, specify the `rollup` modifier as the group and pass the fields you want subtotaled - Type and Lead Source - as parameters. Set `q = group q by rollup('Type', 'LeadSource');`. Here's the full query.

```
q = load "opportunityData";
q = group q by rollup('Type', 'LeadSource');
q = order q by ('Type', 'LeadSource');
q = foreach q generate
    'Type' as 'Type',
    'LeadSource' as 'LeadSource',
    sum('Amount') as 'sum_Amount';
```

The query results show sum of amount by opportunity type and then by lead source. Subtotaled and grand totaled rows have null values for dimensions.

Type	LeadSource	Sum of Amount
Existing Business	Advertisement	6,870,000
	Internet	6,660,000
	Partner	9,500,000
	Trade Show	39,860,000
	Word of mouth	23,400,000
	-	86,290,000
New Business	Advertisement	87,760,000
	Partner	6,750,000
	Trade Show	7,200,000
	Word of mouth	24,310,000
	-	126,020,000
-	-	212,310,000

Example: `grouping()`

Null values in place of labeled totals can confuse query results. Avoid this confusion by labeling totals as `All Types` or `All Lead Sources` using case statements with `grouping()` functions.

```
q = load "opportunityData";
q = group q by rollup('Type', 'LeadSource');
q = order q by ('Type', 'LeadSource');
q = foreach q generate
    (case
        when grouping('Type') == 1 then "All Types"
        else 'Type'
    end) as 'Type',
    (case
        when grouping('LeadSource') == 1 then "All Lead Sources"
        else 'LeadSource'
    end) as 'LeadSource',
    sum('Amount') as 'sum_Amount';
```

Now the query results include labeled totals.

Type	LeadSource	Sum of Amount
Existing Business	Advertisement	6,870,000
	Internet	6,660,000
	Partner	9,500,000
	Trade Show	39,860,000
	Word of mouth	23,400,000
	All Lead Sour...	86,290,000
New Business	Advertisement	87,760,000
	Partner	6,750,000
	Trade Show	7,200,000
	Word of mouth	24,310,000
	All Lead Sour...	126,020,000
All Types	All Lead Sour...	212,310,000

SAQL REFERENCE

These hands-on SAQL examples walk you through writing a query to retrieve data

SAQL Basic Elements
Basic elements are the building blocks of your SAQL query.

SAQL Operators
Use operators to perform mathematical calculations or comparisons.

SAQL Statements
A query is made up of statements. Each SAQL statement has an input stream, an operation, and an output stream.

SAQL Functions
Use functions to perform complex operations on your data.

SAQL Basic Elements

Basic elements are the building blocks of your SAQL query.

Statements
A SAQL query loads input data, operates on it, and outputs the result data. A query is made up of statements. Each SAQL statement has an input stream, an operation, and an output stream.

Keywords
Keywords are case-sensitive and must be lowercase.

Identifiers
SAQL identifiers are case-sensitive and must be enclosed in single quotation marks (').

Number Literals
A number literal represents a number in your script.

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

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

Multivalue Field
A multivalue field contains more than one value.

Quoted String Escape Sequences
Strings can be escaped with the backslash character.

Special Characters
Certain characters have special meanings in SAQL.

Comments

To add a single-line comment in SAQL, preface your comment with two hyphens (`--`). To add a multi-line comment, start your comment with a forward slash and asterisk (`/*`) and end it with an asterisk and forward slash (`*/`).

Statements

A SAQL query loads input data, operates on it, and outputs the result data. 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 is the last line in a query, which doesn't have to be assigned 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 line in this SAQL query is a SAQL statement.

```
q = load "Dataset1";
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';
```

SEE ALSO:

- filter
- foreach
- limit
- offset
- order

Keywords

Keywords are case-sensitive and must be lowercase.

SEE ALSO:

- sample

Identifiers

SAQL identifiers are case-sensitive and must be enclosed in single quotation marks (').

Identifiers that are enclosed in quotation marks can contain any character that a string can contain.

This example uses valid syntax:

```
q = load "Opportunity";

--'Stage' is enclosed in single quotes because it is a field. "08 - Closed Won" is enclosed
```

```
 in double quotes because it is a string.
q = filter q by 'Stage' == "08 - Closed Won";
q = group q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', count() as 'count';
```

This example is **not** valid because you can't use double quotes for an identifier.

```
--this should be 'Account_Owner' in single quotes
q = group q by "Account_Owner";
```

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 "Data";
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.

Multivalue Field

A multivalue field contains more than one value.

Example: One typical use case for multivalue fields is security. For example, you can have a dataset that contains various accounts, and each account has multiple owner IDs. We've created a sample dataset called OppRoles where `OwnerId` is a multivalue field.

Account ID	Amount	Owner.Name	Opportunity ID	Owner ID	Stage
001R00000046CHdIAM	1,900,013	Emily Dickinson	006R0000002OF6eIAG	005R0000000VU9bIAG, 005R0000000VU9bIAH, 005R0000000VU9bIAI	Closed Won

Account ID	Amount	Owner.Name	Opportunity ID	Owner ID	Stage
001R00000046CV6IAM	70,449	Albert Einstein	006R0000002OF6eIAG	005R0000000VU9UIAW, 005R0000000VU9bIAG, 005R0000000VU9UIAX 005R0000000VU9UIAY	Closed Won
001R00000046CI6IAM	4,206,995	Indiana Jones	006R0000002OF6eIAG	006R0000002OF6gIAG, 005R0000000VU9RIAW, 005R0000000VU9SIAW	Closed Won

This query filters on an `OwnerId` to display only the accounts that it can access.

```
q = load "OppRoles";
q = filter q by 'OwnerId' in ["005R0000000VU9bIAG"];
q = foreach q generate 'AccountId' as 'AccountId', 'Amount' as 'Amount', 'Id' as 'Id',
 'Owner.Name' as 'Owner.Name', 'OwnerId' as 'OwnerId', 'StageName' as 'StageName';
```

Warning: When using comparison operators in the filter, use `in` and `not in` to return the correct values. Using `==` and `!=` returns unexpected values when null handling is enabled. See `Group-by` with Null Values for more information.

Account ID	Amount	Owner.Name	Opportunity ID	Owner ID	Stage
001R00000046CHdIAM	1,900,013	Emily Dickinson	006R0000002OF6eIAG	005R0000000VU9bIAG	Closed Won
001R00000046CV6IAM	70,449	Albert Einstein	006R0000002OF6eIAG	005R0000000VU9bIAG	Closed Won

The `OwnerID` value `005R0000000VU9bIAG` has access to two of the three accounts, so two of the accounts are displayed.

Important: Limit multivalue field use to filtering only. Multivalue fields can behave unpredictably with `group` and `foreach`.

SEE ALSO:

- mv_to_string()
- Comparison Operators
- Limit Multivalue Fields

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

Sequence	Meaning
`\'`	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']`.
`::`	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

To add a single-line comment in SAQL, preface your comment with two hyphens (`--`). To add a multi-line comment, start your comment with a forward slash and asterisk (`/*`) and end it with an asterisk and forward slash (`*/`).

Single-Line Comments

Here's an example of a single-line comment on its own line.

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

You can put a comment at the end of a line of SAQL code.

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

To comment a line of SAQL code, add two hyphens at the beginning of the line.

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

Multi-Line Comments

Here's an example of a multi-line comment.

```
q = load "campaign_data";
q = group q by Owner;
q = foreach q generate count() as 'count';
/*
q = limit q 5;
*/
```

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 of the same type. For example, you can compare strings with strings and numbers with numbers.

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

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

Simple case Operator
Use `case` in a `foreach` statement to assign different field values in different situations. `case` supports two syntax forms: searched case and simple case. This section explains simple case.

Searched case Operator
Use `case` in a `foreach` statement to assign different field values in different situations. `case` supports two syntax forms: searched case and simple case. This section shows searched case.

Null Operators

Use `is null` and `is not null` to check whether a value is or is not null. `is null` returns `True` when a value is null. `is not null` returns `True` when a value is not 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 of the same type. For example, you can compare strings with strings and numbers with numbers.

Operator	Name	Description
==	Equals	Returns `True` if the operands are equal. String comparisons that use the equals operator are case-sensitive. Note: For multivalue fields, use `in` to identify rows that contain some value.
!=	Not equals	Returns `True` if the operands aren't equal. Note: For multivalue fields, use `not in` to identify rows that don't contain some value.
<	Less than	Returns `True` if the left operand is less than the right operand.
<=	Less or equal	Returns `True` if the left operand is less than or equal to the right operand.
>	Greater than	Returns `True` if the left operand is greater than the right operand.
>=	Greater or equal	Returns `True` if the left operand is greater than or equal to the right operand.
`like`	Like	Returns `True` if the left operand contains the string on the right. Wildcards and regular expressions aren't supported. This operator is case-sensitive. To match any single character in the string, include an underscore (_). To match any pattern of zero or more characters include a percent sign (%). Starting a pattern with a percent sign returns all words that are either the pattern itself or that end with it. Ending a pattern with a percent sign returns all the words that are

Operator	Name	Description

		either the pattern itself or that begin with it. To match a pattern anywhere in a string, the pattern must start and end with a percent sign.
		To include a literal percent sign or underscore in a pattern, you must escape them with a backwards slash (\).
		This query matches names such as Anita Boyle, Annie Booth, Derek Jernigan, and Hazel Jennings.
		`q = filter q by Customer_Name like "%ni%";`
		This query matches names that begin with "ne" or contain "ne." These names include Andrew Levine, Annette Boone, Annette Cline, and Annie Horne.
		`q = filter q by Customer_Name like "ne%";`
		Use with `!` to exclude records. For example, the following query shows all customer names that don't contain "po."
		`q = filter q by !(Customer_Name like "%po%");`

`matches`	Matches	Returns `True` if the left operand contains the string on the right. Wildcards and regular expressions aren't supported. This operator isn't case-sensitive. Single-character matches aren't supported.
		For example, the following query matches airport codes such as LAX, LAS, ALA, and BLA.
		`my_matches = filter a by origin matches "LA";`
		Use with `!` to exclude records. For example, the following query shows all opportunities where Stage isn't equal to Closed Lost or Closed Won:
		`q = filter q by !('Stage' matches "Closed");`

`in`	In	Returns `True` if the left operand contains 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, the query returns `True` if the left operand is in the array on the right. Use the `date()` function to filter by date ranges. If you search for values in an empty array, in returns `False`. Ranges that are out of order evaluate to `False`. For example, `["Z" .. "A"]` evaluates to `False`.
`not in`	Not in	Returns `True` if the left operand isn't equal to any of the values in an array on the right.

Note: If you use `matches` or `like` with multivalue fields, make sure to use `mv_to_string()` to match all relevant values. If any of the values in the array satisfies the condition, the query returns the first value. The query returns array values in numerical or alphabetical order. The first value it returns in this case isn't necessarily the value that satisfied the condition.

SEE ALSO:

filter

Multivalue Field

Multivalue Field

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

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

x	!x
True	False
False	True
Null	Null

Simple `case` Operator

Use `case` in a `foreach` statement to assign different field values in different situations. `case` supports two syntax forms: searched case and simple case. This section explains simple case.

Syntax

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

`case...end` opens and closes the case operator.

`primary_expr` is any expression that takes a single input value and returns a single output value. May contain values, identifiers, and scalar functions (including date and math functions). The expression can return a number, string, or date.

`when...then` defines a conditional statement. A `case` expression can contain one or more conditional statements.

`test_expr` is any expression that takes a single input value and returns a single output value. May contain values, identifiers, and scalar functions (including date and math functions). The expression must return the same data type as *`primary_expr`*.

`result_expr` is any expression that takes a single input value and returns a single output value. May contain values, identifiers, and scalar functions (including date and math functions). The expression must return the same data type as *`primary_expr`*.

`else default_expr` (optional) is any expression that takes a single input value and returns a single output value. May contain values, identifiers, and scalar functions (including date and math functions). The expression can return a number, string, or date.

Usage

Statements are evaluated in the order that they are given. If *`test_expr`* returns `true`, the corresponding *`result_expr`* is returned. You can specify any number of `when/then` statements.

You can use `else` to specify a default expression. For example, if no industry is specified then use the string "No Industry Specified". If you don't specify a default statement then `null` is returned.

You can use `case` expressions in `foreach` statements. You cannot use `case` in `order`, `group`, or `filter` statements.

Example

Suppose that you want to create a dimension that displays the meaning of industry codes. Use `case` to parse the Industry_Code field and specify the corresponding string.

```
q = foreach q generate Amount as 'Amount', 'Industry_Code' as 'Industry_Code', (case
'Industry_Code'
      when 541611 then "Consulting services"
      when 541800 then "Advertising"
      when 561400 then "Support services"
      else "Unspecified"
end) as 'Industry';
```

The resulting data displays the meaning of industry codes:

Amount	Industry Code	Industry
637,520	541,611	Consulting services
1,750,200	541,800	Advertising
1,935,980	561,400	Support services
4,067,300	541,611	Consulting services
219,000	541,800	Advertising
1,005,200	561,400	Support services

Handling Null Values

In general, `null` values can't be compared. When *`primary_expr`* or *`test_expr`* evaluates to `null`, the *`default_expr`* is returned. If no default expression is specified, `null` is returned.

SEE ALSO:

Speed Up Queries with Dataflow Transformations

Searched `case` Operator

Use `case` in a `foreach` statement to assign different field values in different situations. `case` supports two syntax forms: searched case and simple case. This section shows searched case.

Syntax

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

case...end opens and closes the case operator.

when...then defines a conditional statement. A case expression can contain one or more conditional statements.

search_condition can be any scalar expression that returns a boolean value. It can be a complex boolean expression or a nested case, as long as the result is boolean. For a list of supported operators, see Comparison Operators on page 28.

result_expr is any expression that takes a single input value and returns a single output value. Can contain values, identifiers, and scalar functions (including date and math functions). The expression must return the same data type as specified in the search condition.

else default_expr (optional) is any expression that takes a single input value and returns a single output value. Can contain values, identifiers, and scalar functions (including date and math functions). The expression can return a number, string, or date.

Usage

Statements are evaluated in the order that they are given. If the condition is ***primary_expr*** `==` ***test_expr***, then the corresponding *result_expr* is returned. You can specify any number of `when/then` statements.

You can use `else` to specify a default expression. For example, if no industry is specified, you can use the string "No Industry Specified". If you don't specify a default statement, then `null` is returned.

You can use `case` expressions in `foreach` statements. You cannot use `case` in `order`, `group`, or `filter` statements.

Example

Suppose that you want to see the median deal size for each of your reps. You want to bin their median deal size into the buckets "Small", "Medium", and "Large". Use `case` to assign values to the median deal size.

```
q = load "data";
q = group q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', median('Amount') as 'Median
Amount', (case

  when median('Amount') < 1000000 then "Small"
  when median('Amount') > 1600000 then "Large"
  else "Medium"

end ) as 'Category';
```

The resulting data shows the median deal size for each rep, along with the appropriate bin label.

Account Owner	Category	Median Amount
Bruce Kennedy	Medium	1373900
Catherine Brown	Small	399740
Chris Riley	Medium	1373900
Dennis Howard	Small	517301
Doroth Gardner	Medium	1079956.15
Eric Gutierrez	Small	771320

Handling Null Values

In general, `null` values can't be compared. When the search condition evaluates to `null`, the *`default_expr`* is returned. If no default expression is specified, `null` is returned.

Null Operators

Use `is null` and `is not null` to check whether a value is or is not null. `is null` returns `True` when a value is null. `is not null` returns `True` when a value is not null.

This example returns rows that contain `Sub_Category` fields that are not null and the counts of rows that contain each field.

```
q = load "Superstore";
q = filter q by 'Sub_Category' is not null;
q = group q by 'Sub_Category';
q = foreach q generate 'Sub_Category' as 'Sub_Category', count() as 'count';
q = limit q 2000;
```

Sub-Category	Count of Rows
Accessories	775
Appliances	466
Art	796
Binders	1,523
Bookcases	228
Chairs	617
Copiers	68
Envelopes	254
Fasteners	217

Sub-Category	Count of Rows
Furnishings	957
Labels	364
Machines	115
Paper	1,370
Phones	889
Storage	846
Supplies	190
Tables	319

Replace Null Values with `case`

Use `case` to replace null values with a value of your choice. This example labels the null `Sub-Category` field "Empty."

```
q = load "Superstore";
q = group q by 'Sub_Category';
q = foreach q generate case when 'Sub_Category' is null then "Empty" else 'Sub_Category'
end as 'Sub_Category', count() as 'count';
q = limit q 2000;
```

Sub-Category	Count of Rows
Accessories	775
Appliances	466
Art	796
Binders	1,523
Bookcases	228
Chairs	617
Copiers	68
Envelopes	254
Fasteners	217
Furnishings	957
Labels	364
Machines	115
Paper	1,370
Phones	889

Sub-Category	Count of Rows
Storage	846
Supplies	190
Tables	319
Empty	4

SEE ALSO:

filter

group-by rollup

group-by

SAQL Statements

A query is made up of statements. Each SAQL statement has an input stream, an operation, and an output stream.

cogroup

Use `cogroup` to combine data from two or more data streams into a single data stream. The data streams must have at least one common field.

fill

Use `fill()` to fill in any gaps in date fields. By specifying the date fields to check, `fill()` creates a row that contains the missing month, day, week, quarter, or year and includes a null value. To include values outside the bounds of your data's date range, specify a start date and end date to override existing limits. The function returns the missing date rows with null values.

filter

Selects rows from a dataset based on a filter predicate.

foreach

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

group-by

Organizes the rows returned from a query into groups. Within each group, you can apply an aggregate function, such as `count()` or `sum()` to get the number of items or sum, respectively.

group-by rollup

`rollup` is a subclause of `group-by` that creates and displays aggregations of grouped data. The output of `rollup` is based on column order in your query.

limit

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

load

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

offset

Use `offset` to page through the results of your query.

order

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

sample

Returns a random sample from a large dataset, where each data point has an equal probability of being selected. This keyword uses the Bernoulli distribution.

timeseries

Uses existing data to predict future data points. The `timeseries` statement must follow a projection statement in your query. Perform any filtering pre-projection or after the `timeseries` statement.

union

Combines multiple result sets into one result set. The result sets must have the same field names and structure. You can use a different dataset to create each result set, or you can use the same dataset.

cogroup

Use `cogroup` to combine data from two or more data streams into a single data stream. The data streams must have at least one common field.

`cogroup` is similar to relational database joins, but with some important differences. Unlike a relational database join, in a cogroup the datasets are grouped first, and then the groups are joined. You can use `cogroup` in these ways:

- inner `cogroup`
- left outer `cogroup`
- right outer `cogroup`
- full outer `cogroup`

Note: The statements `cogroup` and `group` are interchangeable. For clarity, we use `group` for statements involving one data stream and `cogroup` for statements involving two or more data streams.

Inner cogroup

Inner `cogroup` combines data from two or more data streams into a resulting data stream. The resulting data stream only contains values that exist in both data streams. That is, unmatched records are dropped.

Syntax

```
result = cogroup data_stream_1 by field1, data_stream_2 by field2;
```

field1 and *field2* must be the same type, but can have different names. For example, `q=group ops by 'Owner', quota by 'Name';`

Example - Inner cogroup

Suppose that you want to understand how much time your reps spend meeting with each account. Is there a relationship between spending more time and winning an account? Are some reps spending much more or much less time than average? To answer these questions, first combine meeting data with account data using `cogroup`.

Suppose that you have a dataset of meeting information from the Salesforce Event object. In this example, your reps have had six meetings with four different companies. The Meetings dataset has a MeetingDuration column, which contains the meeting duration in hours.

#	Company	MeetingDuration
1	Shoes2Go	2
2	FreshMeals	3
3	ZipBikeShare	4
4	Shoes2Go	5
5	FreshMeals	1
6	ZenRetreats	6

The account data exists in the Salesforce Opportunity object. The Ops dataset has an Account, Won, and Amount column. The Amount column contains the dollar value of the opportunity, in millions.

#	Account	Won	Amount
1	Shoes2Go	1	1.5
2	FreshMeals	1	2
3	ZipBikeShare	1	1.1
4	Shoes2Go	0	3
5	FreshMeals	1	1.4
6	ZenRetreats	0	2

To see the effect of meeting duration on opportunities, you start by combining these two datasets into a single data stream using `cogroup`.

```
q = cogroup ops by 'Account', meetings by 'Company';
```

Internally (you cannot see these results yet), the resulting cogrouped data stream contains the following data. Note how the data streams are rolled up on one or more dimensions.

```
(1,{(Shoes2Go,2,), (Shoes2Go,5)},{(Shoes2Go,1,1.5), (Shoes2Go,0,3})

(2,{(FreshMeals,3), (FreshMeals, 5)},{(FreshMeals,1,2) (FreshMeals, 1, 1.4)})

(3,{(ZipBikeShare,4)},{(ZipBikeShare,1, 1.1)})

(4,{(ZenRetreats,6)},{(ZenRetreats,0, 2)})
```

Now the datasets are combined. To see the data, you create a projection using `foreach`:

```
ops = load "Ops";
meetings = load "Meetings";
```

```
q = cogroup ops by 'Account', meetings by 'Company';
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
sum(meetings.'MeetingDuration') as 'TimeSpent';
```

The resulting data stream contains the sum of amount and total meeting time for each company. The sum of amount is the sum of the dollar value for every opportunity for the company.

Account	Sum of Amount	TimeSpent
Company1	4.5	7
Company2	3.4	4
Company3	1.1	4
Company4	2	6

Now that you have combined the data into a single data stream, you can analyze the effects that total meeting time has on your opportunities.

Left Outer `cogroup`

Left outer `cogroup` combines data from the right data stream with the left data stream. The resulting data stream only contains values that exist in the left data stream. If the left data stream has a value that the right data stream does not, the missing value is null in the resulting data stream.

Tip: Use `coalesce` to replace a null value with the value of your choice.

Syntax

```
result = cogroup data_stream_1 by field1 left, data_stream_2 by field2;
```

field1 and *field2* must be the same type, but can have different names. For example, `q=group ops by 'Owner' left, quota by 'Name';`

Example - Left Outer `cogroup` With `coalesce`

Suppose that you want to see what percentage of quota that your reps have obtained. Your quota dataset shows each employee's quota (notice that Farah does not have a quota):

Employee	Quota
Lilly Chow	18,000,000
Emily Dickinson	15,000,000
Jonathan James	17,000,000

Your opportunities data shows the opportunity amount that each employee has won (notice that Jonathan does not have a won opportunity).

Employee	Amount
Lilly Chow	6,000,000
Emily Dickinson	5,000,000
Farah Khan	15,000,000
Lilly Chow	10,000,000
Emily Dickinson	11,000,000

Use a left outer `cogroup` to show only employees that have quotas. Also show the percentage of quota attained.

```
quota = load "Quota";
opp = load "Opportunity";
q = group quota by 'Employee' left, opp by 'Employee';
q = foreach q generate quota.'Employee' as 'Employee',
trunc(sum(opp.'Amount')/sum(quota.'Quota')*100, 2) as 'Percent Attained';
```

Jonathan has not won any opportunities yet, so his percent attained is null.

Employee	Percent Attained
Emily Dickinson	106.66
Jonathan James	-
Lilly Chow	88.88

Use `coalesce` to replace the null opportunities with a zero.

```
quota = load "Quota";
opp = load "Opportunity";
q = group quota by 'Employee' left, opp by 'Employee';
q = foreach q generate quota.'Employee' as 'Employee',
trunc(coalesce(sum(opp.'Amount'),0)/sum(quota.'Quota')*100, 2) as 'Percent Attained';
```

Now Jonathan's percent attained is displayed as zero.

Employee	Percent Attained
Emily Dickinson	106.66
Jonathan James	0
Lilly Chow	88.88

Right Outer `cogroup`

Right outer `cogroup` combines data from the left data stream with the right data stream. The resulting data stream only contains values that exist in the right data stream. If the right data stream has a value that the left data stream does not, the missing value is null in the resulting data stream.

Tip: Use `coalesce` to replace a null value with the value of your choice.

Syntax

```
result = cogroup data_stream_1 by field1 right, data_stream_2 by field2;
```

field1 and *field2* must be the same type, but can have different names. For example, `q=group ops by 'Owner' right, quota by 'Name';`

Full Outer `cogroup`

Full outer `cogroup` combines data from the left and right data streams. The resulting data stream contains all values. If one data stream has a value that the other data stream does not, the missing value is null in the resulting data stream.

Tip: Use `coalesce` to replace a null value with the value of your choice.

Syntax

```
result = cogroup data_stream_1 by field1 full, data_stream_2 by field2;
```

field1 and *field2* must be the same type, but can have different names. For example, `q=group ops by 'Owner' full, quota by 'Name';`

SEE ALSO:

- union
- Combine Data from Multiple Data Streams with cogroup
- Replace Null Values with coalesce()
- group-by

`fill`

Use `fill()` to fill in any gaps in date fields. By specifying the date fields to check, `fill()` creates a row that contains the missing month, day, week, quarter, or year and includes a null value. To include values outside the bounds of your data's date range, specify a start date and end date to override existing limits. The function returns the missing date rows with null values.

Syntax

```
results = fill resultSet by (dateCols=(dateField1, dateField2, "<date format>"),
startDate=startDate, endDate=endDate, [partition])
```

Name	Description
`resultSet`	Required. The results of a query that serve as input to the `fill()` function.

Name	Description
`dateCols`	Required. ***`date_fields`***—The date fields in which to check for gaps. The ***`date format`*** string accepts these values. • `'yearField','monthField', 'Y-M'` • `'yearField', 'quarterField', 'Y-Q'` • `'yearField', 'Y'` • `'yearField', 'weekField', 'Y-W'` • `'yearField', 'monthField', 'dayField', 'Y-M-D'` `startDate`—The starting date value beyond the scope of your data's date range. `endDate`—The ending date value beyond the scope of your data's date range. • You can use `startDate` and `endDate` together or one and not the other. • If you leave out `startDate`, then the start date is the earliest date in your dataset. • If you leave out `endDate`, then the end date is the latest date in your dataset. • If `startDate` and `endDate` are within the bounds of your dataset, `fill()` ignores them.
`partition`	Optional. A field used to split query results into smaller partitions. The `fill()` function resets when the field value changes. After each group of rows is completed for a given partition, `fill()` runs on the next partition.

Example

This example uses `fill()` to add missing quarter and year values to tourist data.

```
q = load "TouristsData";
q = foreach q generate date_Year, date_Quarter, tourists;
q = fill q by (dateCols=(date_Year, date_Quarter, "Y-Q"));
q = limit q 15;
```

The query first returns the year, quarter, and number of tourists for each quarter. Based on the results from the first three years represented in the dataset, the only date data available is for the first quarter.

These are the results from `q = load "TouristsData"; q = foreach q generate date_Year, date_Quarter, tourists;`.

year	quarter	tourists
2001	1	4127

year	quarter	tourists
2002	1	4173
2003	1	4621

`fill()` specifies in the `date_cols` array to group the input data by the quarter and year. To have a complete dataset of years and quarters, `fill()` adds the 2nd, 3rd, and 4th quarters for each year and a null value for the number of tourists.

year	quarter	tourists
2001	1	4127
2001	2	-
2001	3	-
2001	4	-
2002	1	4173
2002	2	-
2002	3	-
2002	4	-
2003	1	4621
2003	2	-
2003	3	-
2003	4	-

Example with Extended Date Range

This query returns null values for tourists where `date_Month` and `date_Year` come before or after the date values in the dataset or there are gaps within the data provided.

```
q = load "TouristsData";
q = foreach q generate date_Year, date_Month, tourists;
q = fill q by (dateCols=(date_Year, date_Month, "Y-M"), startDate="2000-10",
endDate="2001-07");
q = limit q 10;
```

date_Month	date_Year	tourists
10	2000	-
11	2000	-
12	2000	-
01	2001	41,735

date_Month	date_Year	tourists
02	2001	-
03	2001	-
04	2001	26,665
05	2001	-
06	2001	-
07	2001	-

`filter`

Selects rows from a dataset based on a filter 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 query exceed the values supported by SAQL. For example, if something like `filter q by dim1 in {{results(Query_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
- Null Operators
- Use Group and Filter Pre-projection

`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.

Projected Field Names

Each field name in a projection must be unique and not have the name 'none'. Invalid field names throw an error.

For example, the last line in this query is invalid because the same name is used for multiple projected fields:

```
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;
```

The following query is also invalid because the projected field name can't be 'none'.

```
q = load "Products";
q = group q by all;
q = foreach q generate count() as 'none';
q = limit q 2000;
```

SEE ALSO:

Statements

group-by

Organizes the rows returned from a query into groups. Within each group, you can apply an aggregate function, such as `count()` or `sum()` to get the number of items or sum, respectively.

Syntax

`group-by` takes this syntax.

```
group data_stream by fields;
```

data_stream
Data input to group.

fields
Fields by which data is grouped.

Group-by One Field

In this example, the query counts the number of rows for each `Category` field and groups the counts by category.

```
q = load "Superstore";
q = group q by 'Category';
q = foreach q generate 'Category' as 'Category', count() as 'count';
q = limit q 2000;
```

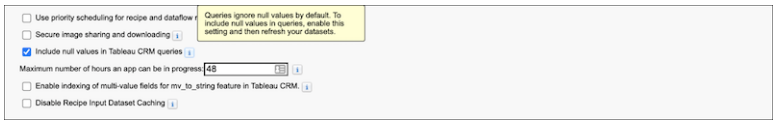
Category	Count of Rows
Furniture	2,121
Office Supplies	6,026
Technology	1,847

Note: `cogroup` and `group-by` are interchangeable. For clarity, we use `group-by` for statements that involve one data stream and `cogroup` for statements that involve two or more data streams.

`Group-by` with Null Values

To return grouped null values in your queries, you must select the preference to include null values in Setup. Otherwise, queries ignore null values.

1. In Setup, enter *`Analytics`* in the Quick Find box.
2. Select **Settings** from the list of Analytics options.
3. In **Settings**, click the checkbox for **Include null values in Analytics queries**.



Here's an example of a query that returns null values. It orders the results by the `Sub_Category` field and specifies that the results display in ascending order, with nulls first.

```
q = load "Superstore";
q = group q by 'Sub_Category';
q = foreach q generate 'Sub_Category' as 'Sub_Category', count() as 'count';
q = order q by 'Sub_Category' asc nulls first;
q = limit q 2000;
```

Sub-Category	Count of Rows
-	4
Accessories	775
Appliances	466
Art	796
Binders	1,523
Bookcases	228
Chairs	617
Copiers	68
Envelopes	254
Fasteners	217
Furnishings	957
Labels	364
Machines	115
Paper	1,370

Sub-Category	Count of Rows
Phones	889
Storage	846
Supplies	190
Tables	319

SEE ALSO:

- Aggregate Functions
- Null Operators
- cogroup
- Use Group and Filter Pre-projection

group-by rollup

`rollup` is a subclause of `group-by` that creates and displays aggregations of grouped data. The output of `rollup` is based on column order in your query.

Syntax

`group-by rollup` takes this syntax.

```
group data_stream by rollup(fields);
```

data_stream
Data input to group.

fields
Fields by which data is grouped.

Note: `rollup` works with `group-by` only. You cannot use it with `cogroup`.

`rollup` supports the following aggregate functions.

- `average()`
- `count()`
- `min()`
- `max()`
- `sum()`
- `unique()`

This example first groups the results by `Category` and `Sub-Category`, and runs `sum('Sales')`, an aggregate function on each resulting row. By modifying the `group-by` clause with `rollup`, the query "rolls up" the results into subtotals and a grand total.

```
q = load "Superstore";
q = group q by rollup('Category', 'Sub_Category');
q = order q by ('Category');
```

```
q = foreach q generate 'Category' as 'Category', 'Sub_Category' as 'Sub_Category',
sum('Sales') as 'sum_sales';
```

Category	Sub-Category	sum_sales
Furniture	Bookcases	114,348
	Chairs	328,237
	Furnishings	91,514
	Tables	206,966
	-	741,064
Office Supplies	Appliances	107,532
	Art	27,119
	Binders	203,413
	Envelopes	16,363
	Fasteners	3,024
	Labels	12,486
	Paper	78,479
	Storage	223,844
	Supplies	46,674
	-	718,934
Technology	Accessories	167,380
	Copiers	149,528
	Machines	189,239
	Phones	329,636
	-	835,783
	-	2,295,781

The query first groups the total sales for each sub-category of a given category. Next, it groups the total sales for a single category. After each category's total sales is accounted for, the query generates the total sales for all categories.

`rollup` with Null Values

Note: To return grouped null values in your queries, you must select the null handling for dimensions preference in Setup. See `group-by` for more information.

This example shows how null values display in query results. The query is the same as the one in the first example.

```
q = load "Superstore";
q = group q by rollup('Category', 'Sub_Category');
q = foreach q generate 'Category' as 'Category', 'Sub_Category' as 'Sub_Category',
sum('Sales') as 'sum_sales';
q = order q by ('Category', 'Sub_Category');
```

Category	Sub-Category	sum_sales
Furniture	Bookcases	114,348
	Chairs	328,237
	Furnishings	91,514
	Tables	206,966
	-	92
	-	741,156
Office Supplies	Appliances	107,532
	Art	27,119
	Binders	203,413
	Envelopes	16,363
	Fasteners	3,024
	Labels	12,486
	Paper	78,479
	Storage	223,844
	Supplies	46,674
	-	273
	-	719,206
Technology	Accessories	167,380
	Copiers	149,528
	Machines	189,239
	Phones	329,636
	-	259
	-	836,041
-	Computers	113
	Projectors	744
	-	562

Category	Sub-Category	sum_sales
		1,420
		2,297,824

The query first groups the total sales for each sub-category of a given category. In this example, each category contains a null sub-category. The value of the null sub-category is also included in the total sales for each sub-category.

After the query accounts for all of the named categories—categories that have a value—it displays the sub-categories and total sales for null categories. Finally, the query generates the total sales for all categories.

`rollup` with Null Values and `case` Statements

Use the `grouping` function and `case` statements together to label the subtotal and grand total categories. In this example, the first case checks for a null value generated by the rollup in the `Category` field. If true, then the query labels the field `All Categories`. The second case checks whether a `Sub-Category` field is similarly null. If true, then the query labels the field `All Sub-Categories`.

```
q = load "Superstore";
q = group q by rollup ('Category', 'Sub_Category');
q = foreach q generate
(case
    when grouping('Category') == 1 then "All Categories"
    else 'Category'
end) as 'Category',
(case
    when grouping('Sub_Category') == 1 then "All Sub-Categories"
    else 'Sub_Category'
end) as 'SubCategory', sum('Sales') as 'sum_sales';
```

Category	Sub-Category	sum_sales
Furniture	Bookcases	114,348
	Chairs	328,237
	Furnishings	91,514
	Tables	206,966
	-	92
	All Sub-Categories	741,156
Office Supplies	Appliances	107,532
	Art	27,119
	Binders	203,413
	Envelopes	16,363
	Fasteners	3,024

Category	Sub-Category	sum_sales
	Labels	12,486
	Paper	78,479
	Storage	223,844
	Supplies	46,674
	-	273
	All Sub-Categories	719,206
Technology	Accessories	167,380
	Copiers	149,528
	Machines	189,239
	Phones	329,636
	-	259
	All Sub-Categories	836,041
-	Computers	113
	Projectors	744
	-	562
	All Sub-Categories	1,420
All Categories	All Sub-Categories	2,297,824

SEE ALSO:

- Null Operators
- Simple case Operator
- Aggregate Functions
- grouping()

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

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 not grouped. The columns are the columns of the loaded dataset.

Example: Load the Accounts dataset to the stream 'b'. `b = load "Accounts";`

offset

Use `offset` to page through the results of your query.

Syntax

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

Usage

Skips over the specified number of rows when returning the results of a query. You typically use `offset` to paginate the query results.

When using `offset` in your SAQL statements, be aware of these rules:

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

Example - Return Rows 51–101

This example loads the opportunity dataset, sorts the rows in alphabetical order by account owner, and returns rows 51–101:

```
q = load "DTC_Opportunity";
q = order q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', 'Account_Type' as 'Account_Type',
 'Amount' as 'Amount';
q = offset q 50;
q = limit q 50;
```

SEE ALSO:

Statements

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 ]);
result = order rows by field [ asc | desc ] nulls [first | last];
```

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

Usage

Use `order` to sort the results in a data stream for display. You can use `order` with ungrouped data. You can also use `order` to sort grouped data by an aggregated value.

Do not use `order` to specify the order that another SAQL statement or function will process records in. For example, do not use `order` before `timeseries` to change the order of processing. Instead, use `timeseries` parameters.

By default, nulls are sorted last when sorting in ascending order and first when sorting in descending order. You can change the ordering of nulls using nulls `[first | last]`.

Note: Applying labels to dimension values in the XMD changes the displayed values, but doesn't change the sort order.

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: By default, nulls are sorted first when sorting in descending order. To change the null sort order to last, use this syntax:

```
q = order q by last_shipping_cost desc nulls last;
```

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

sample

Returns a random sample from a large dataset, where each data point has an equal probability of being selected. This keyword uses the Bernoulli distribution.

Syntax

```
sample(percentage-size-of-dataset) repeatable(seed)
```

sample
Required. Specifies the percentage of the dataset that is returned as a random sample. The percentage size value can be any positive decimal.

repeatable
Optional. To create a random sample deterministically, specify a seed. `sample` returns the same subset of data each time you pass repeatable the same seed value. The seed value can be any positive integer.

Usage

Use `sample` to project a query on a representative sample from your dataset, where each data point has an equal probability of being selected. `sample` runs pre-projection.

Add `sample` and `repeatable` after the `load` statement. Any operation performed on the query after the `load` statement affects only the random sample of data. Let's look at an example.

```
q = load "Opportunity" sample(10) repeatable(1);
q = group q by all;
q = foreach q generate count() as 'count';
q = limit q 2000;
```

Count of Rows
453

Here, the query returns the row count of the sample, 453—around 10% of the dataset's 4.6k rows. The `repeatable` keyword guarantees that the query always returns the same result. Without the `repeatable` keyword, the query returns a sample of a slightly different size each time you run it. If you modify your dataset and add more data, then `repeatable` doesn't return the same result.

`group-by` Example

This query returns the counts of opportunities for each stage. Since the query operates on 10% of the dataset, the counts for each stage are approximately 1/10 of the original count.

```
q = load "Opportunity" sample(10) repeatable(1);
q = group q by 'StageName';
q = foreach q generate 'StageName', count() as 'count';
q = limit q 2000;
```

Stage	Count of Rows
Closed Lost	89
Closed Won	254
Id. Decision Makers	13
Needs Analysis	15

Stage	Count of Rows
Negotiation/Review	6
Perception Analysis	13
Proposal/Price Quote	9
Prospecting	10
Qualification	25
Value Proposition	19

`filter` Example

This query returns only the won opportunities for each stage. Since the query operates on 10% of the dataset, the count for each stage is approximately 1/10 of the original count.

```
q = load "Opportunity" sample(10);
q = filter q by 'IsWon' == "true";
q = group q by 'StageName';
q = foreach q generate 'StageName', count() as 'count';
q = limit q 2000;
```

Stage	Count of Rows
Closed Won	275

SEE ALSO:

Keywords

`timeseries`

Uses existing data to predict future data points. The `timeseries` statement must follow a projection statement in your query. Perform any filtering pre-projection or after the `timeseries` statement.

Usage

`timeseries` crunches your data and selects the forecasting model that gives the best fit. You can let `timeseries` select the best model or specify the model you want. `timeseries` detects seasonality in your data. It considers periodic cycles when predicting what your data will look like in the future. You can specify the type of seasonality or let `timeseries` choose the best fit.

The amount of data, which is required to make a prediction depends on how your data is filtered and grouped. For example, for a non-seasonal monthly model, 2 data points are sufficient, whereas for a seasonal monthly model, at least 24 data points (two seasonal cycles) are required. If you don't have enough data to make a good prediction, `timeseries` returns nulls in the data. If no data is passed to `timeseries`, an empty dataset is returned.

Syntax

```
result = timeseries resultSet generate (measure1 as fmeasure1 [, measure2 as
fmeasure2...]) with (parameters);
```

measure1, *measure2* and so on are the measures that you want to predict future values for. You can predict measures from grouping queries or from simple values queries. The predicted values and the original values are projected together. The columns from the previous `foreach` statement are also projected.

parameters can have the following values:

- `length` (required) Number of points to predict. For example, if length is 6 and the `dateCols` type string is `Y-M`, `timeseries` predicts data for 6 months.

  Note: If you want to use `dateCols` but your data stream has missing dates, use `fill` before using `timeseries`.

  `timeseries` makes the most accurate prediction possible by choosing the best algorithm for your data. Predictive algorithms are more accurate for shorter time periods.

- `dateCols` (optional) Date fields to use for grouping the data, plus the date column type string. For example, `dateCols=(CloseDate_Year, CloseDate_Month, "Y-M")`. Date columns are projected automatically. Allowed values are:
  - ***YearField, MonthField,*** `"Y-M"`
  - ***YearField, QuarterField,*** `"Y-Q"`
  - ***YearField,*** `"Y"`
  - ***YearField, MonthField, DayField*** `"Y-M-D"`
  - ***YearField, WeekField*** `"Y-W"`
- `ignoreLast` (optional) If `true`, `timeseries` doesn't use the last time period in the calculations. The default is `false`.

  Set this parameter to `true` to improve the accuracy of the forecast if the last time period contains incomplete data. For example, if you're partway through the quarter, `timeseries` forecasts more accurately if you set this parameter to `true`.
- `order` (optional) Specify the field to use for ordering the data. Mandatory if `dateCols` isn't used. By default, this field is sorted in ascending order. Use `desc` to specify descending order, for example `order=('Type' desc)`. You can also order by multiple fields, for example `order=('Type' desc, 'Group' asc)`.

  For example, suppose that your data has no date columns, but it has a measure column called Week. Use `order='Week'`.

  Note: Specify either `dateCols` or `order`.
- `partition` (optional) Specify the column used to partition the data. The column must be a dimension. The `timeseries` calculation is done separately for each partition to ensure that each partition uses the most accurate algorithm. For example, data in one partition might have a seasonal variation while data in another partition doesn't. The partition columns are projected automatically.

  For example, suppose that your sales data for raw materials contains the date sold, type of raw material, and the weight sold. To predict the future weight sold for each type of raw material, use `partition='Type'`.
- `predictionInterval` (optional) Specify the uncertainty, or confidence interval, to display at each point. Allowed values are `80` and `95`. The upper and lower bounds of the confidence interval are projected in columns named ***column_name***`_low_95` and ***column_name***`_high_95`.
- `model` (optional) Specify which prediction model to use. If unspecified, `timeseries` calculates the prediction for each model and selects the best model using Bayesian information criterion (BIC).

Allowed values are:

- `None` `timeseries` selects the best algorithm for the data
- `Additive` uses Holt's Linear Trend or Holt-Winters method with additive components.
- `Multiplicative` uses Holt's Linear Trend or Holt-Winters method with multiplicative components

- `seasonality` (optional) Use with `dateCols` to specify the seasonality for your prediction. Allowed values are:
  - `0` No seasonality
  - any integer between `2` and `24`

  If unspecified, `timeseries` calculates the prediction for each type of seasonality and select the results with the smallest error.

  Example

`seasonality`	`dateCols`	**Type of Seasonality**
`seasonality=4`	`dateCols="Y-Q"`	Yearly seasonality, because there are four quarters in a year.
`seasonality=12`	`dateCols="Y-M"`	Yearly seasonality, because there are 12 months in a year.
`seasonality=7`	`dateCols="Y-M-D"`	Weekly seasonality, because there are seven days in a week.

Tips

Here's how you can make the most of using `timeseries`:

- Are you currently part way through the month, quarter, or year? Consider setting `ignoreLast` to `true` so that `timeseries` doesn't use the partial data in the current time period, leading to a more accurate prediction.
- Is `timeseries` not returning any data? If there aren't enough data points to make a good prediction, `timeseries` returns `null`. Try increasing the number of data points.
- Is `timeseries` returning an error? You could have gaps in your dates or times. Like all good forecasting algorithms, `timeseries` needs a continuous set of dates with no gaps, including in each partition. If you think your data has date gaps, try using `fill` first.

Example - How Many Tourists Will Visit Next Year?

Suppose that you run a chain of retail stores, and the number of tourists in your city affect your sales. Use `timeseries` to predict how many tourists will come to your city next year:

```
q = load "TouristData";
q = group q by ('Visit_Year', 'Visit_Month');
q = foreach q generate 'Visit_Year', 'Visit_Month', sum('NumTourist') as 'sum_NumTourist';

-- If your data is missing some dates, use fill() before using timeseries()
-- Make sure that the dateCols parameter in fill() matches the dateCols parameter in
timerseries()
q = fill q by (dateCols=('Visit_Year','Visit_Month', "Y-M"));

-- Use timeseries() to predict the number of tourists.
```

```
q = timeseries q generate 'sum_NumTourist' as Tourists with (length=12,
dateCols=('Visit_Year','Visit_Month', "Y-M"));

q = foreach q generate 'Visit_Year' + "~~~" + 'Visit_Month' as 'Visit_Year~~~Visit_Month',
 Tourists;
```

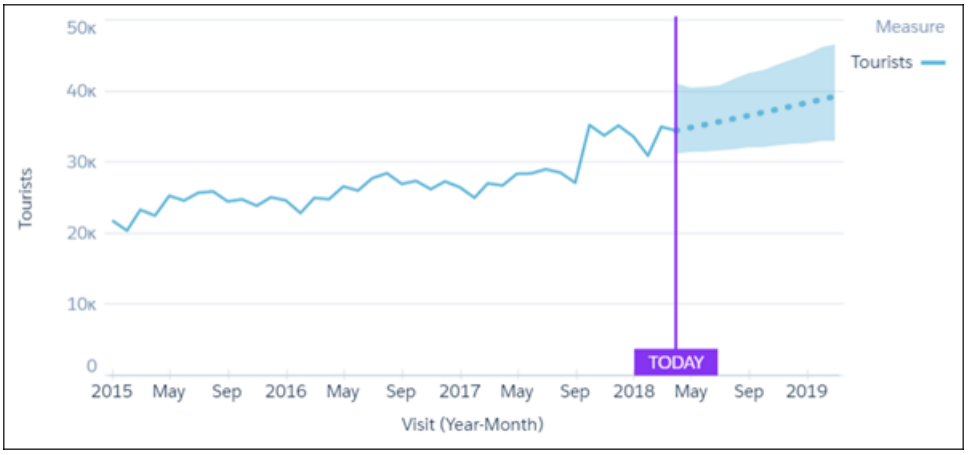
Use a timeline chart and set a predictive line to see the calculated future data. The resulting graph shows the likely number of tourists in the future.

Example - Predict a Range With 95% Accuracy

Suppose that you wanted to predict the number of tourists in your city next year with 95% accuracy. Use `predictionInterval=95` to set a 95% confidence interval for the number of tourists. The upper and lower bounds are projected as the fields `Tourists_high_95` and `Tourists_low_95`.

```
q = load "TouristData";
q = group q by ('Visit_Year', 'Visit_Month');
q = foreach q generate 'Visit_Year', 'Visit_Month', sum('NumTourist') as 'sum_NumTourist';

-- If your data is missing some dates, use fill() before using timeseries()
-- Make sure that the dateCols parameter in fill() matches the dateCols parameter in
timerseries()
q = fill q by (dateCols=('Visit_Year','Visit_Month', "Y-M"));

-- use timeseries() to predict the number of tourists
q = timeseries q generate 'sum_NumTourist' as 'fTourists' with (length=12,
predictionInterval=95, dateCols=('Visit_Year','Visit_Month', "Y-M"));
q = foreach q generate 'Visit_Year' + "~~~" + 'Visit_Month' as 'Visit_Year~~~Visit_Month',
 coalesce(sum_NumTourist,fTourists) as 'Tourists', fTourists_high_95, fTourists_low_95;
```

Use a timeline chart and set a predictive line to see the calculated future data. In the timeline chart options, select Single Axis for the **Axis Mode**, fTourists_high_95 for **Measure 1**, and fTourists_low_95 for **Measure 2**. The resulting graph shows the likely number of tourists in the future and the 95% confidence interval.

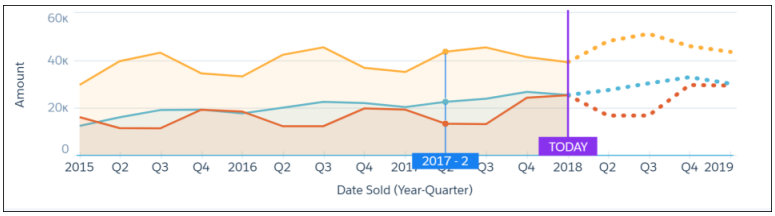


Example - Predict Seasonal Data

Suppose that you want to predict the revenue for each type of account. You know that your account revenue has yearly seasonality and that you want to group dates by quarter, so you specify `dateCols=('Date_Sold_Year','Date_Sold_Quarter', "Y-Q")` and `seasonality = 4`. To see the predicted values over the next year, use `length=4` to specify four quarters.

```
q = load "Account";
q = group q by ('Date_Sold_Year', 'Date_Sold_Quarter', 'Type');
q = foreach q generate 'Date_Sold_Year', 'Date_Sold_Quarter', 'Type', sum('Amount') as
'sum_Amount';

-- If your data is missing some dates, use fill() before using timeseries()
-- Make sure that the dateCols parameter in fill() matches the dateCols parameter in
timerseries()
q = fill q by (dateCols=('Date_Sold_Year','Date_Sold_Quarter', "Y-Q"));

-- use timeseries() to predict the amount sold
q = timeseries q generate 'sum_Amount' as Amount with (partition='Type',length=4,
dateCols=('Date_Sold_Year','Date_Sold_Quarter', "Y-Q"), seasonality = 4);
q = foreach q generate 'Date_Sold_Year' + "~~~" + 'Date_Sold_Quarter' as
'Date_Sold_Year~~~Date_Sold_Quarter','Type', Amount ;
```

Use a timeline chart and set a predictive line to see the calculated future data. The resulting graph shows the likely sum of revenue for each account, taking into account the quarterly seasonal variation.



SEE ALSO:

Forecast Future Data Points with timeseries

union

Combines multiple result sets into one result set. The result sets must have the same field names and structure. You can use a different dataset to create each result set, or you can use the same dataset.

Syntax

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

Example

```
q = union q1, q2, q3;
```

Example

You want to see how each rep compares to the average for deals won. You can make this comparison by appending these two result sets together:

- Total amount of opportunities won for each rep
- Average amount of opportunities won for all reps

Then use `union` to append the two result sets.

First, show the total amount of won opportunities for each rep.

```
opt = load "DTC_Opportunity_SAMPLE";
opt = filter opt by 'Won' == "true";

-- group by owner
rep = group opt by 'Account_Owner';

-- project the sum of amount for each rep
rep = foreach rep generate 'Account_Owner' as 'Account_Owner', sum('Amount') as 'sum_Amount';

rep = order rep by 'sum_Amount' asc;
```

The resulting graph shows the sum of amount for each rep.

Account Owner	Sum of Amount
Laura Garza	31,605,866
Doroth Gardner	29,543,120
Johnny Green	25,672,424
Irene Kelley	25,308,421

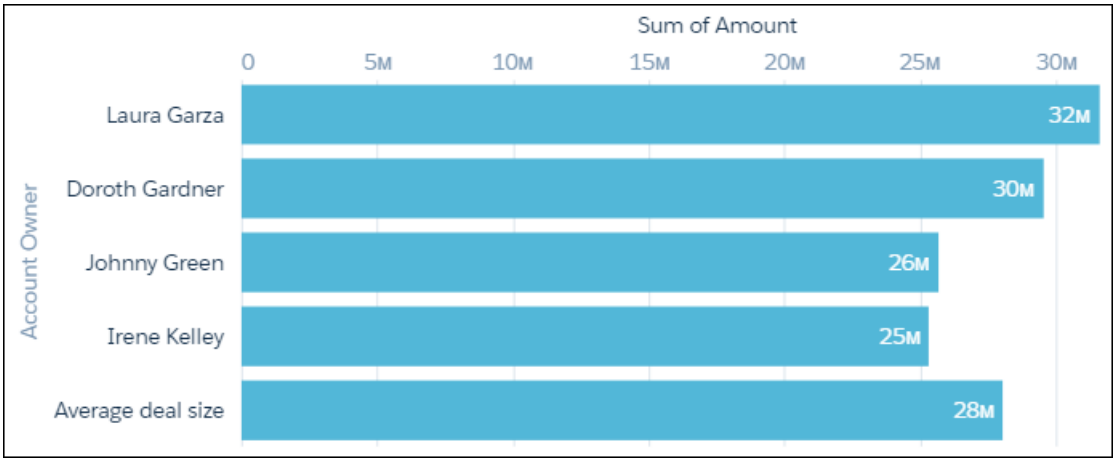
Next, calculate the average of the sum of the amounts for each rep using the `average` function.

```
-- grouping rep by all returns all the data in a single row.
avg_rep = group rep by all;

-- Calculate the average of the Sum of Amount column.
-- Use the text 'Average Deal Size' in the 'Account Owner' column
avg_rep = foreach avg_rep generate "Average deal size" as 'Account_Owner', avg('sum_Amount')
 as 'sum_Amount';
```

Because the two data streams have the same field names and structure, you can use `union` to combine them.

```
q = union rep, avg_rep;
```

The resulting graph contains the sum of amounts by each rep together with the average amount per rep.



Combine the SAQL fragments to get the complete SAQL statement.

```
opt = load "DTC_Opportunity_SAMPLE";
opt = filter opt by 'Won' == "true";

-- group by owner
rep = group opt by 'Account_Owner';

-- project the sum of amount for each rep
rep = foreach rep generate 'Account_Owner' as 'Account_Owner', sum('Amount') as 'sum_Amount';

rep = order rep by 'sum_Amount' desc;

-- grouping rep by all returns all the data in a single row.
avg_rep = group rep by all;


-- Calculate the average of the Sum of Amount column.
-- Use the text ‘Average Deal Size’ in the ‘Account Owner’ column
avg_rep = foreach avg_rep generate "Average deal size" as 'Account_Owner', avg('sum_Amount')
 as 'sum_Amount';

q = union rep, avg_rep;
```

SEE ALSO:

- cogroup
- Append Datasets using union

SAQL Functions

Use functions to perform complex operations on your data.

- Aggregate Functions
  Aggregate functions perform computations across all values of a grouped field.

Date Functions

Use SAQL date functions to perform time-based analysis.

Time Zone Date Functions

When you enable the time zone feature, you can use the fields of the `DateTime` and `DateOnly` type to access date information in the specified time zone. For example, if a user in New York runs a SAQL query, they see date information displayed in Eastern Standard time.

Work with Custom Fiscal Year Data

After inheriting custom fiscal years, SAQL queries support custom fiscal year data.

String Functions

Use SAQL string functions to format your measure and dimension fields.

Math Functions

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

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.

coalesce

Use `coalesce()` to get the first non-null value from a list of parameters, or to replace nulls with a different value.

Aggregate Functions

Aggregate functions perform computations across all values of a grouped field.

If you don't precede an aggregate function by a `group by` statement, it treats each line as its own group. Using an aggregate function on an empty set returns null.

avg() or average()

Returns the average of the values of a measure field.

count()

Returns the number of rows that match the query criteria.

first()

Returns the first value for the specified field.

last()

Returns the last value in the tuple for the specified field.

max()

Returns the maximum value of a measure field.

median()

Returns the median value of a measure field.

min()

Returns the minimum value of a measure field.

sum()

Returns the sum of a numeric field.

unique()

Returns the count of unique values.

stddev()

Returns the standard deviation of the values in a field. Accepts measure fields (but not expressions) as input.

stddevp()

Returns the population standard deviation of the values in a field. Accepts measure fields as input but not expressions.

var()

Returns the variance of the values in a field. Accepts measure fields as input but not expressions.

varp()

Returns the variance of the values in a field. Accepts measure fields as input but not expressions.

percentile_cont()

Calculates a percentile based on a continuous distribution of the column value.

percentile_disc()

Returns the value corresponding to the specified percentile.

regr_intercept()

Uses two numerical fields to calculate a trend line, then returns the y-intercept value. Use this function to find out the likely value of *field_y* when *field_x* is zero.

regr_slope()

Uses two numerical fields to calculate a trend line, then returns the slope. Use this function to learn more about the relationship between two numerical fields.

regr_r2()

Uses two numerical fields to calculate R-squared, or goodness of fit. Use `regr_r2()` to understand how well the trend line fits your data.

grouping()

Returns 1 if null dimension values are due to higher-level aggregates (which usually means the row is a subtotal or grand total), otherwise returns 0.

SEE ALSO:

group-by

`avg()` or `average()`

Returns the average of the values of a measure field.

Example - Calculate the Average Amount of an Opportunity Grouped by Type

Use `avg()` to compare the average size of opportunities for each account type.

```
q = load "DTC_Opportunity";
q = group q by 'Account_Type';
q = foreach q generate 'Account_Type' as 'Account_Type', avg('Amount') as 'avg_Amount';
```

SEE ALSO:

median()

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';
```

`count()` operates on the stream that is input to the `group` or `cogroup` statement. It doesn't operate on the newly grouped stream or on an ungrouped stream.

```
q = load "Carriers";
q = group q by (Year);
q = foreach a1 generate count(q) as countYear, count() as count, Year as year;
```

first()

Returns the first value for the specified field.

Use `first()` to return the first value of a measure or dimension. You can also use `first()` used to return the value of a field without grouping by that field.

Note: If the values are not sorted, the 'first' value could be any value in the tuple.

Example - Return the First Industry for an Account Owner

Your reps own opportunities in several industries. You need a list of rep names with their **first** industry, where industry is sorted alphabetically. Group by account owner and industry, sort by industry, then use `first()` to get the first industry.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by ('Account_Owner', 'Industry');
q = foreach q generate 'Account_Owner' as 'Account_Owner', 'Industry' as 'Industry';
q = order q by 'Industry';

q = foreach q generate 'Account_Owner' as 'Account_Owner', first('Industry') as 'One
Industry';
```

Account Owner	One Industry
Bruce Kennedy	Agriculture
Chris Riley	Agriculture
Dennis Howard	Agriculture
Eric Gutierrez	Agriculture
Eric Sanchez	Agriculture
Evelyn Williamson	Agriculture

Example - Return Any Industry for an Account Owner

Your reps own opportunities in several industries. You need a list of rep names with any **one** of a rep's industry - it doesn't matter which one. In this case. Group by account owner then use `first()` to get the first industry from an unsorted collection.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', first('Industry') as 'One
Industry';
```

The resulting table displays each rep along with **one** of their industries (basically the first industry from an unsorted collection).

Account Owner	One Industry
Bruce Kennedy	Agriculture
Catherine Brown	Engineering
Chris Riley	Agriculture
Dennis Howard	Healthcare
Doroth Gardner	Utilities
Eric Gutierrez	Education

SEE ALSO:

last()

`last()`

Returns the last value in the tuple for the specified field.

Use `last()` to return the last value of a measure or dimension. You can also use `last()` used to return the value of a field without grouping by that field.

Note: If the values are not sorted, the 'last' value could be any value in the tuple.

SEE ALSO:

first()

`max()`

Returns the maximum value of a measure field.

Example - Find the Largest Opportunity for Each Account

```
q = load "Ops";
q = group q by 'Account_Name';
q = foreach q generate 'Company' as 'Company', max('Amount') as 'Largest Deal';
```

SEE ALSO:

min()

median()

Returns the median value of a measure field.

Example - Find the Median Time to Close a Case

Use `median()` to find the median amount of time it takes to resolve a case, grouped by company.

```
q = load "Case";
q = group q by 'Account_Name';
q = foreach q generate 'Account_Name' as 'Account_Name', median('CallDuration') as
'median_CallDuration';
q = order q by 'Account_Name' asc;
```

SEE ALSO:

avg() or average()

min()

Returns the minimum value of a measure field.

Example - Find the Smallest Opportunity For Each Account

```
q = load "Ops";
q = group q by 'Account_Name';
q = foreach q generate 'Company' as 'Company', min('Amount') as 'Smallest Deal';
```

SEE ALSO:

max()

sum()

Returns the sum of a numeric field.

Example - Calculate the Total Meeting Time

Suppose that you have a database of meeting information. Use `sum()` to see that the total time spent meeting with each account.

```
q = load "Meetings";
q = group q by 'Company';
q = foreach q generate 'Company' as 'Company', sum('MeetingDuration') as 'sum_meetings';
```

`unique()`

Returns the count of unique values.

Example - Count the Number of Industries

Use `unique()` to count the number of different industries that you have opportunities with.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by all;
q = foreach q generate unique('Industry') as 'unique_Industry';
```

`stddev()`

Returns the standard deviation of the values in a field. Accepts measure fields (but not expressions) as input.

Example - Look at Variability in Amount

Use `stddev()` to get a feel for the amount of spread, or dispersion, in the size of your deals.

```
q = load "DTCOpps";
q = group q by all;
q = foreach q generate stddev('Amount') as 'stddev_Amount';
```

Should I Use `stddev()` or `stddevp()`?

Use `stddev()` when the values in your field are a partial sample of the entire set of values (that is, a partial sampling of the whole population). Use `stddevp()` when your field contains the complete set of values (that is, the entire population of values).

SEE ALSO:

stddevp()

`stddevp()`

Returns the population standard deviation of the values in a field. Accepts measure fields as input but not expressions.

Example - Calculate the Population Standard Deviation of Amount

Use `stddevp()` to calculate the population standard deviation of the amount of each opportunity. Group by product family to see which type of product has the greatest variability in deal size.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by 'Product_Family';
```

```
q = foreach q generate 'Product_Family' as 'Product_Family', stddevp('Amount') as
'stddevp_Amount';
```

SEE ALSO:

stddev()

var()

Returns the variance of the values in a field. Accepts measure fields as input but not expressions.

Example - Calculate the Variance of Deal Amount

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by all;
q = foreach q generate var('Amount') as 'var_Amount';
```

SEE ALSO:

varp()

varp()

Returns the variance of the values in a field. Accepts measure fields as input but not expressions.

Example - Calculate the Population Variance of Deal Amount

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by all;
q = foreach q generate varp('Amount') as 'var_Amount';
```

SEE ALSO:

var()

percentile_cont()

Calculates a percentile based on a continuous distribution of the column value.

```
percentile_cont(p) within group (order by expr [asc | desc])
```

`percentile_cont()` accepts a numeric grouped expression *expr* and sorts it in the specified order. If order is not specified, the default order is ascending. It returns the value behind which (100*p)% of values in the group 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.

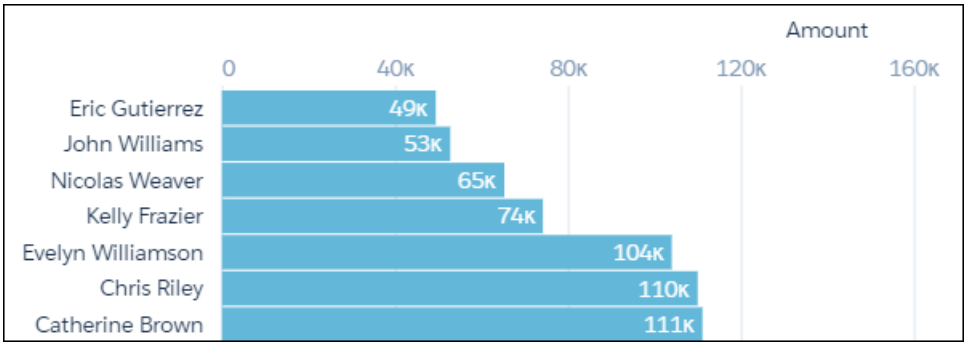
If *expr* contains no value that falls exactly at the 100*p-th percentile mark, `percentile_cont()` returns a value interpolated from the two closest values in *expr*.

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 - Display the Interpolated Value of the Bottom 15% of Deals

Suppose that you want to see the bottom 15% of deals for each rep. You don't need to see the actual deal size - just the 'average' size of the bottom 15%. Use `percentile_cont(.15)`.



SEE ALSO:

percentile_disc()

`percentile_disc()`

Returns the value corresponding to the specified percentile.

```
percentile_disc(p) within group (order by expr [asc | desc])
```

`percentile_disc()` accepts a numeric grouped expression *`expr`* and sorts it in the specified order. If order is not specified, the default order is ascending. It returns the value behind which (100*p)% of values in the group 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()` returns the next value from *`expr`* in the sort order.

For example, if `Mea1` contains the values `[54, 35, 15, 15, 76, 87, 78]` then:

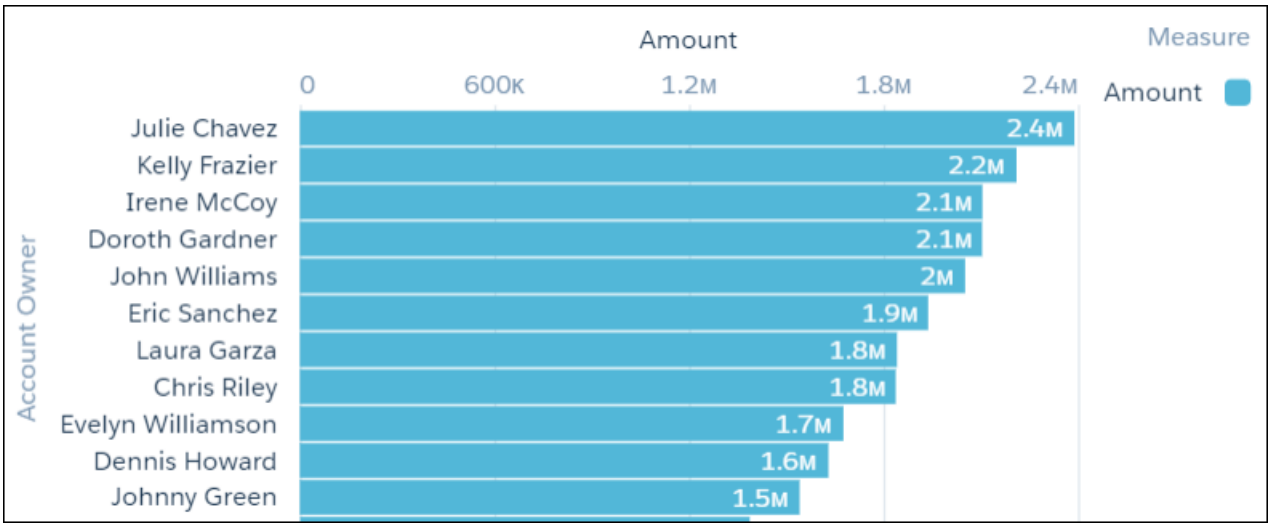
```
percentile_disc(0.5) within group (order by Mea1) == 54
percentile_disc(0.72) within group (order by Mea1) == 78
```

Example - Rank Your Reps by Top Quartile of Deal Size

Suppose that you want to see which reps close the biggest deals. (The result may be different than the sum of deal amount, if some reps close a lot of smaller deals). You also want the chart to display the size of actual deals, not an average of deal size. Use `percentile_disc(.25)` to look at the top quarter of the deal size for each rep.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', percentile_disc(0.25) within
group (order by 'Amount' desc) as 'Amount';
q = order q by 'Amount' desc;
```

You can see that 25% of Julie Chavez's deals are bigger than $2.4 million, and 25% of Kelly Frazier's deals are bigger than $2.2 million. You also know that Julie closed a deal worth$2.4 million, and that number isn't an average.



SEE ALSO:

- percentile_cont()
- Show the Top and Bottom Quartile

regr_intercept()

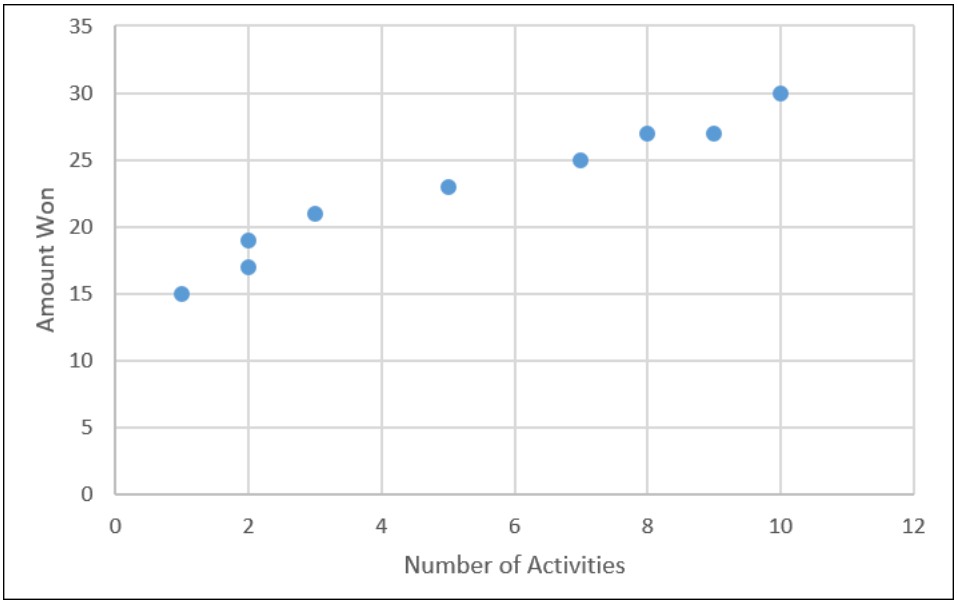
Uses two numerical fields to calculate a trend line, then returns the y-intercept value. Use this function to find out the likely value of *field_y* when *field_x* is zero.

```
regr_intercept(field_y, field_x)
```

field_y is a grouped dependent numeric expression and *field_x* is a grouped independent numeric expression. `regr_intercept(field_y, field_x)` uses simple linear regression to calculate the trend line. The input fields (*field_y*, *field_x*) must contain at least two pairs of non-null values. This function works with simple grouped values but not with cogroups.

Example - What Is the Likely Amount Won If the Number of Activities Is Zero?

Suppose that you have a dataset that includes the number of activities (such as meetings) and the won opportunity amount.



What size of deal can you expect to win if you don't have any activities with an account? `regr_intercept` performs a linear analysis on your data then calculates the y-intercept (that is, the value of Amount Won when Number of Activities is zero).

```
q = load "Data";
q = group q by all;

--trunc() truncates the result to two decimal places
q = foreach q generate trunc(regr_intercept('Amount', 'NumActivities'),2) as intercept;
```

The projected deal size with no activities is $15.04 million dollars.

SEE ALSO:

regr_slope()

regr_slope()

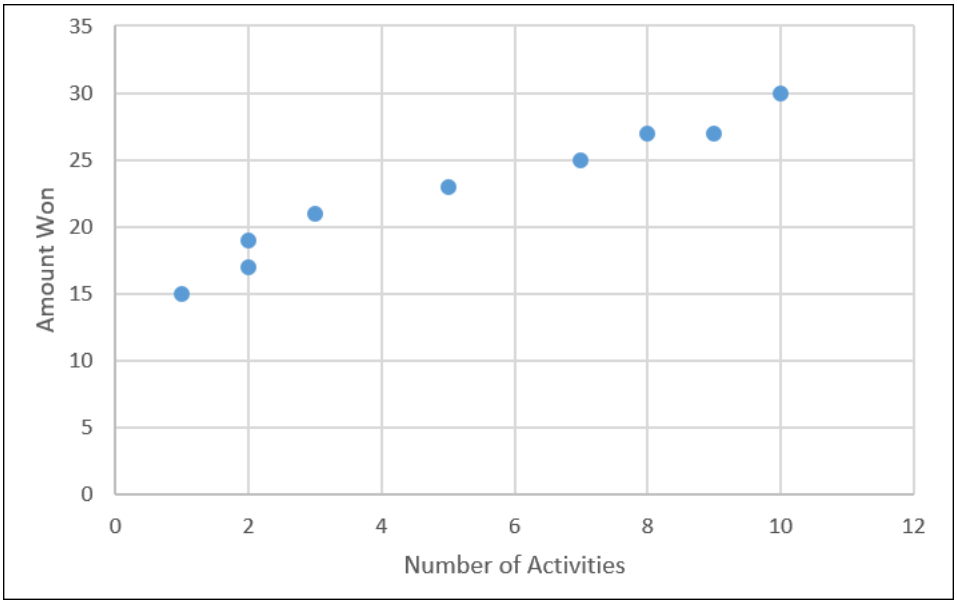
Uses two numerical fields to calculate a trend line, then returns the slope. Use this function to learn more about the relationship between two numerical fields.

```
regr_slope(field_y, field_x)
```

field_y is a grouped dependent numeric expression and *field_x* is a grouped independent numeric expression. `regr_slope(field_y, field_x)` uses simple linear regression to calculate the trend line. The input fields (*field_y*, *field_x*) must contain at least two pairs of non-null values. This function works with simple grouped values but not with cogroups.

Example - Calculate the Relationship Between Number of Activities and Deal Amount

Suppose that you have a dataset that includes the number of activities (such as meetings) and the won opportunity amount.



How much bigger with the deal size be for each extra activity? `regr_slope` performs a linear analysis on your data then calculates the slope (that is, the increased amount you win for each extra activity).

```
q = load "data/sales";
q = group q by all;

--trunc() truncates the result to two decimal places
q = foreach q generate trunc(regr_slope('Amount', 'NumActivities'),2) as 'Gain per Activity';
```

Based on your existing data, every extra activity that you have tends to increase the deal size by $1.45 million, on average.

Gain per Activity	1.45

SEE ALSO:

- regr_intercept()
- Calculate the Slope of the Regression Line

regr_r2()

Uses two numerical fields to calculate R-squared, or goodness of fit. Use `regr_r2()` to understand how well the trend line fits your data.

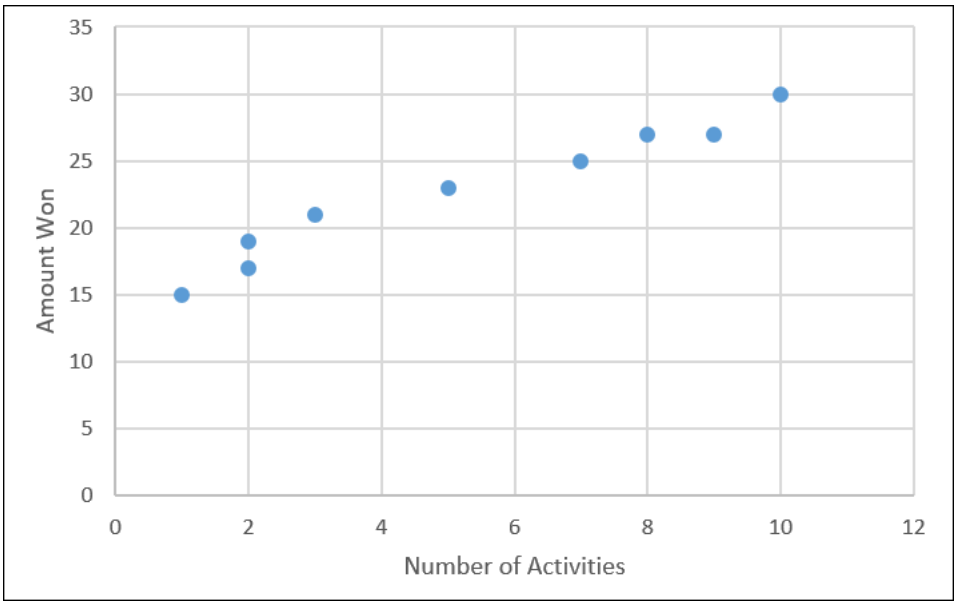
```
regr_r2(field_y, field_x)
```

`field_y` is a grouped dependent numeric expression and *`field_x`* is a grouped independent numeric expression. `regr_r2(field_y, field_x)` uses simple linear regression to calculate a trend line, then calculates R-squared. If the returned value is small, then functions like `regr_slope()` and `regr_intercept()` are likely to return accurate results.

The input fields (*`field_y`*, *`field_x`*) must contain at least two pairs of non-null values. This function works with simple grouped values but not with cogroups.

Example - How Well Does the Calculated Trend Line Fit My Data

Suppose that you have a dataset that includes the number of activities (such as meetings) and the won opportunity amount.



You want to check the calculated trend line for 'goodness of fit' to see how accurate the results from other statistical functions are.

```
q = load "regression";
q = group q by all;

q = foreach q generate trunc(regr_r2('Amount', 'NumActivities'),2) as 'R Squared';
```

The value of R squared is 0.95.

`grouping()`

Returns 1 if null dimension values are due to higher-level aggregates (which usually means the row is a subtotal or grand total), otherwise returns 0.

The `grouping()` function is most useful when paired with the `rollup` modifier on the `group` statement. Invoking `grouping()` lets work with subtotaled data.

Example - Label Subtotaled Data

Suppose that you have a dataset of opportunity information with amounts totaled by lead source and type. Calculate totals with `rollup`. Then use `grouping()` with a `case` statement to check whether a row is a total and if it is then label it as "all" values.

```
q = load "opportunityData";

--Modify the group statement with rollup to calculate subtotals of grouped measures
q = group q by rollup('Type', 'LeadSource');

q = order q by ('Type', 'LeadSource');
```

```
--Determine which rows are totals with grouping(), which returns 1 if a row is a total
q = foreach q generate
    (case
        when grouping('Type') == 1 then "All Types"
        else 'Type'
    end) as 'Type',
    (case
        when grouping('LeadSource') == 1 then "All Lead Sources"
        else 'LeadSource'
    end) as 'LeadSource',
    sum('Amount') as 'sum_Amount';
```

Type	LeadSource	Sum of Amount
Existing Business	Advertisement	6,870,000
	Internet	6,660,000
	Partner	9,500,000
	Trade Show	39,860,000
	Word of mouth	23,400,000
	All Lead Sour...	86,290,000
New Business	Advertisement	87,760,000
	Partner	6,750,000
	Trade Show	7,200,000
	Word of mouth	24,310,000
	All Lead Sour...	126,020,000
All Types	All Lead Sour...	212,310,000

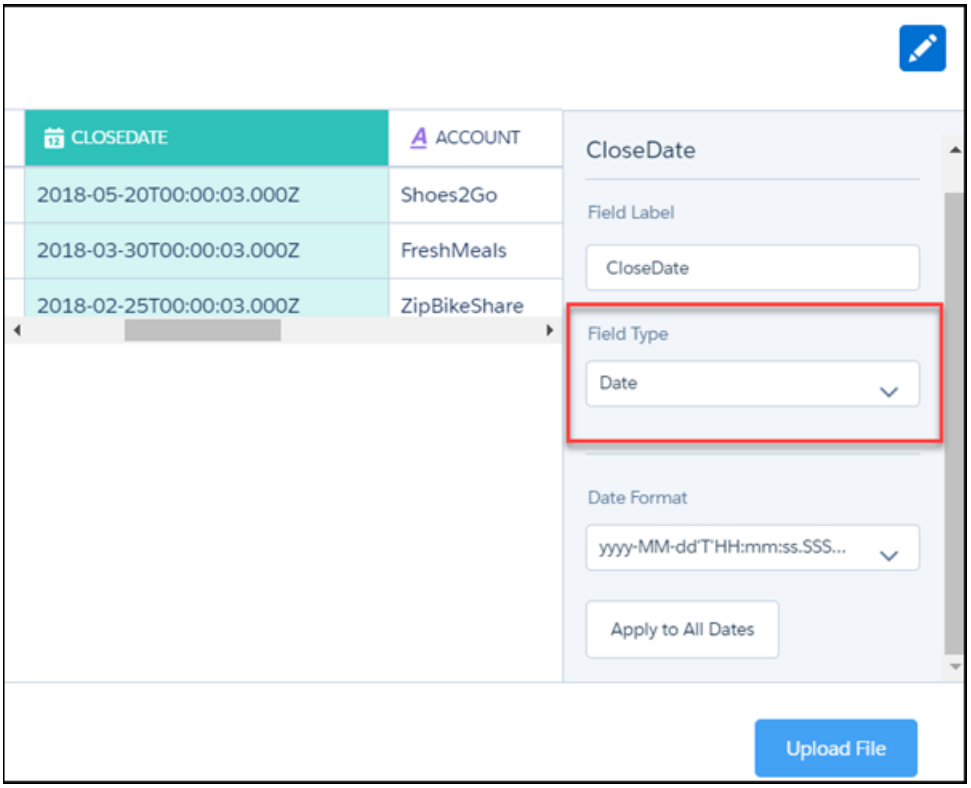
Date Functions

Use SAQL date functions to perform time-based analysis.

Understanding How Date Information is Uploaded to Analytics

When you upload a date field to Analytics, it creates dimension and measure fields to contain the date and time information. You can use SAQL date functions to convert the dimensions and measures to dates. You can then use the dates to sort, filter, and group data in your SAQL queries.

For example, suppose that you upload a dataset that contains the CloseDate date field.



During the dataflow processing, Analytics creates these fields. All the fields are dimensions, except for the epoch fields, which are measures.

Field	Description
CloseDate	A dimension containing the date and time. For example, 2018-02-25T00:00:03.000Z. You can't use this string in a date filter. Instead, 'cast' it to a date type using `toDate()`.
CloseDate (Day)	Dimension containing the day in the month, for example 30.
CloseDate (Hour)	Dimension containing the hour, for example, 11. If the original date did not contain the hour, this field contains 00.
CloseDate (Minute)	Dimension containing the minute, for example, 59. If the original date did not contain the minute, this field contains 00
CloseDate (Month)	Dimension containing the month, for example, 12.
CloseDate(Quarter)	Dimension containing the quarter, for example, 4.
CloseDate (Second)	Dimension containing the second, for example, 59. If the original date did not contain the second, this field contains 00.
CloseDate (Week)	Dimension containing the week, for example, 52.
CloseDate_day_epoch	Measure containing the UNIX epoch time, which is the number of days that have elapsed since 00:00:00, Thursday, 1 January 1970.
CloseDate_sec_epoch	Measure containing the Unix epoch time in seconds. Seconds epoch time is the number of seconds that have elapsed since 00:00:00, Thursday, 1 January 1970.

Analytics creates fields ending in `_Fiscal` for dates associated with a custom fiscal year. Querying dates with this field works the same way as it does for standard fiscal years.

daysBetween()
Returns the number of days between two dates. This function is only valid in a `foreach` statement.

date_diff()
Returns the amount of time between two dates. This function is only valid in a `foreach` statement.

now()
Returns the current datetime in UTC. This function is only valid in a `foreach` statement.

date()
Returns a date that can be used in a filter. This function takes a year, a month, and a day dimension as input.

toDate()
Converts a string or Unix epoch seconds to a date. Returns a date that can be used in another function such as `daysBetween()`. The returned date cannot be used in a filter.

date_to_epoch()
Converts a date to Unix epoch seconds.

date_to_string()
Converts a date to a string.

toString()
Converts a date to a string.

Time-Based Filtering
SAQL gives you many ways to specify the range of dates that you want to look at, such as "all ops from the last fiscal quarter" or "all cases from the last seven days".

Day in the Week, Month, Quarter, or Year
Returns the day in the specified time period for a given date. These functions answer questions like "do we close more deals at the beginning or end of a quarter?".

First Day in the Week, Month, Quarter, or Year
Returns the date of the first day in the specified week, month, quarter, or year.

Last Day in the Week, Month, Quarter, or Year
Returns the date of the last day in the specified week, month, quarter, or year.

Number of Days in the Month, Quarter, or Year
Returns the number of days in the month, quarter, or year for the specified date.

SEE ALSO:
Analyze Your Data Over Time

daysBetween()

Returns the number of days between two dates. This function is only valid in a `foreach` statement.

Syntax

`daysBetween(`***`date1, date2`***`)`

`date1` specifies the start date.

`date2` specifies the end date.

Usage

If *`date1`* is after *`date2`*, the number of days returned is a negative number.

You must use `daysBetween()` in a `foreach()` statement. You cannot use this function in `group by`, `order by`, or `filter` statements.

Example

How many days did it take to close each opportunity? Use `daysBetween()`.

```
q = load "DTC_Opportunity";
q = foreach q generate daysBetween(toDate(Created_Date_sec_epoch),
toDate(Close_Date_sec_epoch) ) as 'Days to Close';
q = order q by 'Days to Close';
```

Example

How long has each opportunity been open for, in days? Use `daysBetween()` and `now()`.

```
q = load "DTC_Opportunity";
q = filter q by 'Closed' == "false";
q = foreach q generate daysBetween(toDate(Created_Date_sec_epoch), now() ) as 'Days to
Close';
q = order q by 'Days to Close';
```

SEE ALSO:

- date_diff()
- Calculate How Long Activities Take

`date_diff()`

Returns the amount of time between two dates. This function is only valid in a `foreach` statement.

Syntax

`date_diff(`***`datepart,startdate,enddate`***`)`

`datepart` specifies how you want to measure the time interval:

- `year`
- `month`
- `quarter`
- `day`

- `week`
- `hour`
- `minute`
- `second`

`startdate` specifies the start date.

`enddate` specifies the end date.

Usage

Returns the time difference between two dates in years, months, or days. For example,

`date_diff("year", toDate("31-12-2015", "dd-MM-yyyy"), toDate("1-1-2016", "dd-MM-yyyy"))` returns 1.

If *`startdate`* is after *`enddate`*, the difference is returned as a negative number.

You must use `date_diff()` in a `foreach()` statement. You cannot use this function in `group by`, `order by`, or `filter` statements.

The maximum amount of time returned is 9,223,372,036,854,775,807 nanoseconds. This maximum amount of time can be measured in any supported *`datepart`* value (nanoseconds aren't supported). For example, in days, the maximum amount of time returned is 106,751.99 days (excluding leap seconds).

Example - How Many Weeks Did Each Opportunity Take to Close?

Use `date_diff()` with *`datepart`* = `week` to calculate how long, in weeks, it took to close each opportunity.

```
q = load "DTC_Opportunity";
q = foreach q generate date_diff("week", toDate(Created_Date_sec_epoch),
toDate(Close_Date_sec_epoch) ) as 'Weeks to Close';
q = order q by 'Weeks to Close';
```

Example - How Long Ago Was an Opportunity Closed?

Use `date_diff()` with *`datepart`* = `month` to calculate how many months have passed since each opportunity closed. Use `now()` as the end date.

```
q = load "DTC_Opportunity";
q = foreach q generate date_diff("month", toDate(Close_Date_sec_epoch), now() ) as 'Months
 Since Close';
q = order q by 'Months Since Close';
```

SEE ALSO:

daysBetween()

now()

Calculate How Long Activities Take

now()

Returns the current datetime in UTC. This function is only valid in a `foreach` statement.

Syntax

`now()`

Usage

This function is commonly used with `daysBetween()`, `date_diff()`, and `date_to_string()`.

Example

How long ago was each opportunity created, in weeks? Use `date_diff()`, *`datepart`* = `week`, and `now()`.

```
q = load "DTC_Opportunity";
q = foreach q generate date_diff("week", toDate(Created_Date_sec_epoch), now() ) as 'Weeks
 to Close';
q = order q by 'Weeks to Close';
```

Example

What is the date today? Use `now()` inside `date_to_string()`.

```
q = load "DTC_Opportunity";

-- Notice how the ' character is escaped with the \ character in 'Today\'s
q = foreach q generate date_to_string(now(), "yyyy-MM-dd") as 'Today\'s Date';
```

SEE ALSO:

date_diff()

`date()`

Returns a date that can be used in a filter. This function takes a year, a month, and a day dimension as input.

Syntax

`date(`***`year, month, day`***`)`

Usage

Specify the year, month, and day. For example:

```
date('OrderDate_Year', 'OrderDate_Month', 'OrderDate_Day')
```

Example

Which opportunities have your reps closed in the past 30 days? Use `date()` to select records with a close date in the past 30 days.

```
q = load "DTC_Opportunity";

-- use date() to create a date that you can use in a filter
-- 'Close_Date_Year', 'Close_Date_Month', and 'Close_Date_Day' are date fields in the
```

```
DTC_Opportunity data set

q = filter q by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["30 days
 ago".."current day"];
q = group q by 'Account_Owner';
q = foreach q generate 'Account_Owner' as 'Account_Owner', sum('Amount') as 'sum_Amount';
q = order q by 'Account_Owner' asc;
```

SEE ALSO:

- toDate()
- Time-Based Filtering

toDate()

Converts a string or Unix epoch seconds to a date. Returns a date that can be used in another function such as `daysBetween()`. The returned date cannot be used in a filter.

Syntax

`toDate(`***`string`*** `[,`***`formatString`***`])`

If a *`formatString`* argument isn't provided, the function uses the format `yyyy-MM-dd HH:mm:ss`

`toDate(`***`epoch_seconds`***`)`

Note: Be sure to use the sec_epoch field and not the day_epoch field.

Example: Display the Number of Days Since an Opportunity Opened

Suppose that you have an opportunity dataset with the account name and the epoch seconds fields:

Account	OrderDate_sec_epoch
Shoes2Go	1,521,504,003
FreshMeals	1,521,158,403
ZipBikeShare	1,518,739,203

You want to see how many days ago an opportunity was opened. Use `daysBetween()` and `now()`. Use `toDate()` to convert the order date epoch seconds to a date format that can be passed to `daysBetween()`.

```
q = load "OpsDates1";

q = foreach q generate Account, daysBetween(toDate(OrderDate_sec_epoch), now()) as
'daysOpened';
```

The resulting data stream displays the number of days since the opportunity was opened.

Account	daysOpened
Shoes2Go	66
FreshMeals	70
ZipBikeShare	98

Note: Because dates are sorted lexicographically, changing the date format also changes the sort order.

SEE ALSO:

date()

date_to_epoch()

Converts a date to Unix epoch seconds.

Syntax

`date_to_epoch(`***`date`***`)`

date_to_string()

Converts a date to a string.

Syntax

`date_to_string(`***`date, formatString`***`)`

Note: This function is identical to `toString()`.

Usage

This function must take a `toDate()` or `now()` function as its first argument.

Example

```
q = foreach q generate date_to_string(now(), "yyyy-MM-dd HH:mm:ss") as ds1;
```

toString()

Converts a date to a string.

Syntax

`toString(`***`date, formatString`***`)`

Note: This function is identical to `date_to_string()`.

Usage

This function must take a `toDate()` or `now()` function as its first argument.

Example

```
q = foreach q generate toString(now(), "yyyy-MM-dd HH:mm:ss") as ds1;
```

Time-Based Filtering

SAQL gives you many ways to specify the range of dates that you want to look at, such as "all ops from the last fiscal quarter" or "all cases from the last seven days".

Using Date Ranges in Filters

Use these filters to specify the date range you want to look at:

- Fixed date range, for example between August 1, 2018 and June 2, 2017
- Relative date range, for example between two years ago and last month
- Open-ended ranges, for example before 04/2/2018
- Add and subtract dates, for example all records from three months before yesterday

Example: Display Opportunities Closed This Month

Suppose that you want to see which opportunities closed this month. Your data includes the account name, the close date fields, and the epoch seconds field.

Account	CloseDate (Year)	CloseDate (Month)	CloseDate_sec_epoch	CloseDate (Day)
Shoes2Go	2018	05	1,526,774,403	20
FreshMeals	2018	03	1,522,368,003	30
ZipBikeShare	2018	02	1,519,516,803	25

Use `date()` to generate the close date in date format. Then use relative date ranges to filter opportunities closed in the current month.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
month" .. "current month"];
q = foreach q generate Account;
```

If the query is run in May 2018, the resulting data stream contains one entry:

Account
Shoes2Go

To add the close date in a readable format, use `toDate()`.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
```

```
month" .. "current month"];
q = foreach q generate Account, toDate('CloseDate_sec_epoch') as 'Close Date';
```

The resulting data stream includes the full date and time of the close date.

Account	Close Date
Shoes2Go	2018-05-20 00:00:03

You can also display just the month and day of the close date.

```
q = load "OpsDates1";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in ["current
month" .. "current month"];
q = foreach q generate Account, 'CloseDate_Month' + "/" + 'CloseDate_Day' as 'Close Date';
```

The resulting data stream contains the month and day of the close date.

Account	Close Date
Shoes2Go	05/20

Fixed Date Ranges

Use `dateRange()` to specify a fixed range of dates in a filter:

```
dateRange(startArray_y_m_d, endArray_y_m_d)
```

`startArray_y_m_d` is an array that specifies the start date

`endArray_y_m_d` is an array that specifies the end date

For example, return all records between October 2, 2014 and August 16, 2016:

```
q = filter q by date('Created_Date_Year', 'Created_Date_Month', 'Created_Date_Day') in
[dateRange([2014,10,2], [2016,8,16])];
```

Relative Date Ranges

Use relative date ranges to answer questions such as "how many opportunities did each rep close in the past fiscal quarter"? To specify a relative date range, use the `in` operator on an array with relative date keywords. For example, return all records from one year ago up to and including the current year.

```
q = filter q  by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["1 year
 ago".."current year"];
```

Return all records from two quarters ago, up to and including two quarters from now.

```
q = filter q  by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["2
quarters ago".."2 quarters ahead"];
```

Return all records from the last two fiscal years, up to and including today.

```
q = filter q by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["2
fiscal_years ago".."current day"];
```

Use these relative date keywords:

- 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

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

Open-Ended Date Ranges

Use open-ended date ranges for queries such as "List all opportunities closed after 12/23/2014". For example, return all records up to and including the current month.

```
q = filter q  by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in [.."1
year ago"];
```

You can also specify a closed relative date range. For example, return all records from three years ago up to and including today.

```
q = filter q by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["3 years
 ago"..];
```

Add and Subtract Dates

You can add and subtract dates using the relative date keywords. For example, return all records from one year ago, up to and including today.

```
q = filter q  by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["current
 day - 1 year"..];
```

Return all records from today up to two years and three months from now.

```
q = filter q by date('Close_Date_Year', 'Close_Date_Month', 'Close_Date_Day') in ["current
 day".."2 years ahead + 3 months"];
```

SEE ALSO:

- date()
- Display the Opportunities Closed This Month

Day in the Week, Month, Quarter, or Year

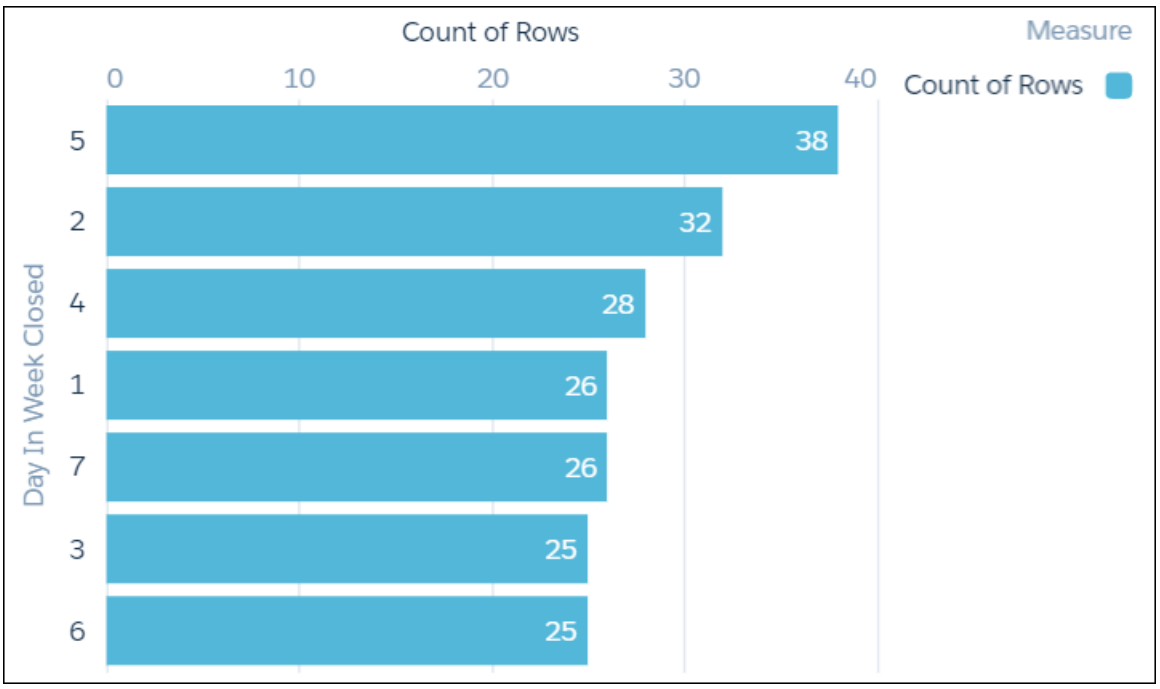
Returns the day in the specified time period for a given date. These functions answer questions like "do we close more deals at the beginning or end of a quarter?".

Example

Suppose that you want to see on which day of the week most deals are closed. Use `day_in_week(date)`.

```
q = load "Data";

q = foreach q generate day_in_week(toDate('Close_Date_sec_epoch')) as 'Day In Week Closed';

q = group q by 'Day In Week Closed';
q = foreach q generate 'Day In Week Closed' as 'Day In Week Closed', count() as 'count';
q = order q by 'count' desc;
```

The resulting data displays the number of opportunities closed, grouped by the day of the week that the opportunities were closed on.



It looks like most opportunities are closed on Thursday (day 5).

day_in_week(date)

Returns an integer representing the day of the week for a specific date. For example, 1 = Sunday, 2 = Monday.

```
q = foreach q generate day_in_week(toDate('Close_Date_sec_epoch')) as 'Day In Week Closed';
```

day_in_month(date)

Returns an integer representing the day of the month for a specific date.

```
q = foreach q generate day_in_month(toDate('Close_Date_sec_epoch')) as 'Day in Month
Closed';
```

day_in_quarter(date)

Returns an integer representing the day of the quarter for a specific date.

```
q = foreach q generate day_in_quarter(toDate('Close_Date_sec_epoch')) as 'Day in Quarter
Closed';
```

day_in_year(date)

Returns an integer representing the day of the year for a specific date.

```
q = foreach q generate day_in_year(toDate('Close_Date_sec_epoch')) as 'Day in Year Closed';
```

First Day in the Week, Month, Quarter, or Year

Returns the date of the first day in the specified week, month, quarter, or year.

Usage

Use these functions in a `foreach()` statement. You cannot use them in `group by`, `order by`, or `filter` statements.

Use the functions whose names begin with `week`, `month`, `quarter`, and `year` with standard calendar year dates. Use the functions whose names begin with `fiscal` with fiscal year dates.

Note: You can't use fiscal date functions in recipes and dataflow transformations.

week_first_day(date)

Returns the date of the first day of the week for the specified date.

```
q = foreach q generate week_first_day(toDate('Close_Date_sec_epoch')) as 'Week First Day';
```

Note: This function always counts the `firstDayOfWeek` as 0 (Sunday). It overrides the `firstDayOfWeek` parameter for `sfdcDigestTransformation` and CSV uploads.

fiscal_week_first_day(date)

Returns the fiscal date of the first day of the week for the specified date.

```
q = foreach q generate fiscal_week_first_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Week First Day';
```

Note: This function respects the `firstDayOfWeek` parameter for `sfdcDigestTransformation` and CSV uploads. The default value is 0 (Sunday).

month_first_day(date)

Returns the date of the first day of the month for the specified date.

```
q = foreach q generate month_first_day(toDate('Close_Date_sec_epoch')) as 'Month First
Day';
```

fiscal_month_first_day(date)

Returns the fiscal date of the first day of the month for the specified date.

```
q = foreach q generate fiscal_month_first_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Month First Day';
```

quarter_first_day(date)

Returns the date of the first day of the quarter for the specified date.

```
q = foreach q generate quarter_first_day(toDate('Close_Date_sec_epoch')) as 'Quarter First
 Day';
```

fiscal_quarter_first_day(date)

Returns the fiscal date of the first day of the quarter for the specified date.

```
q = foreach q generate fiscal_quarter_first_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
 Quarter First Day';
```

year_first_day(date)

Returns the date of the first day of the year for the specified date.

```
q = foreach q generate year_first_day(toDate('Close_Date_sec_epoch')) as 'Year First day';
```

Note: This function always returns 1st January.

`fiscal_year_first_day(date)`

Returns the fiscal date of the first day of the year for the specified date.

```
q = foreach q generate fiscal_year_first_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Year First Day';
```

SEE ALSO:

Date Formats and Fiscal Dates for Source Data

Last Day in the Week, Month, Quarter, or Year

Returns the date of the last day in the specified week, month, quarter, or year.

Usage

Use these functions in a `foreach()` statement. You cannot use them in `group by`, `order by`, or `filter` statements.

Use the functions whose names begin with `week`, `month`, `quarter`, and `year` with standard calendar year dates. Use the functions whose names begin with `fiscal` with fiscal year dates.

Note: You can't use fiscal date functions in recipes and dataflow transformations.

`week_last_day(date)`

Returns the date of the last day of the week for the specified date.

Note: This function always counts the `firstDayOfWeek` as 0 (Sunday). It overrides the `firstDayOfWeek` parameter for `sfdcDigestTransformation` and CSV uploads.

```
q = foreach q generate week_last_day(toDate('Close_Date_sec_epoch')) as 'Week Last Day';
```

`fiscal_week_last_day(date)`

Returns the fiscal date of the last day of the week for the specified date.

Note: This function respects the `firstDayOfWeek` parameter for `sfdcDigestTransformation` and CSV uploads. The default value is 0 (Sunday).

```
q = foreach q generate fiscal_week_last_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Week Last Day';
```

`month_last_day(date)`

Returns the date of the last day of the month for the specified date.

```
q = foreach q generate month_last_day(toDate('Close_Date_sec_epoch')) as 'Month Last Day';
```

fiscal_month_last_day(date)

Returns the fiscal date of the last day of the month for the specified date.

```
q = foreach q generate fiscal_month_last_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Month Last Day';
```

quarter_last_day(date)

Returns the date of the last day of the quarter for the specified date.

```
q = foreach q generate quarter_last_day(toDate('Close_Date_sec_epoch')) as 'Quarter Last
Day';
```

fiscal_quarter_last_day(date)

Returns the fiscal date of the last day of the quarter for the specified date.

```
q = foreach q generate fiscal_quarter_last_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
 Quarter Last Day';
```

year_last_day(date)

Returns the date of the last day of the year for the specified date.

```
q = foreach q generate year_last_day(toDate('Close_Date_sec_epoch')) as 'Year Last Day';
```

Note: This function always returns 31st December. You can use it to find the number of days to the year end.

fiscal_year_last_day(date)

Returns the fiscal date of the last day of the year for the specified date.

```
q = foreach q generate fiscal_year_last_day(toDate('Close_Date_sec_epoch')) as 'Fiscal
Year Last Day';
```

SEE ALSO:

Date Formats and Fiscal Dates for Source Data

Number of Days in the Month, Quarter, or Year

Returns the number of days in the month, quarter, or year for the specified date.

month_days(date)

Returns the number of days in the month for the specified date.

```
q = foreach q generate month_days(toDate('Close_Date_sec_epoch')) as 'Billing Days In
Month';
```

quarter_days(date)

Returns the number of days in the quarter for the specified date.

```
q = foreach q generate quarter_days(toDate('Close_Date_sec_epoch')) as 'Billing Days In
Quarter;
```

year_days(date)

Returns the number of days in the year for the specified date.

```
q = foreach q generate year_days(toDate('Close_Date_sec_epoch')) as 'Billing Days In Year;
```

Time Zone Date Functions

When you enable the time zone feature, you can use the fields of the `DateTime` and `DateOnly` type to access date information in the specified time zone. For example, if a user in New York runs a SAQL query, they see date information displayed in Eastern Standard time.

Note: In SAQL, the `DateOnly` type displays the date with an empty timestamp, for example, "2014-12-31 00:00:00." The inclusion of the timestamp is a limitation of the beta release.

Use Time Zone-Enabled Dates in SAQL Projections
You can project an exact date such as 2017-3-31 23:59:59 or part of a date such as year, month, or day.

Access Date Functions with Time Zone Enabled
Use these functions to get the day, week, year, and other parts of `DateTime` or `DateOnly` fields. The return values are numbers.

Group By Date
You can group the result of your SAQL query by `DateTime` and `DateOnly` fields.

Order By Date
You can order the result of your SAQL queries by `DateTime` or `DateOnly`.

Filter By Date
You can filter results by `DateTime` and `DateOnly` fields. Filters can include exact dates, specific date ranges, or relative date ranges.

Calculate the Time Between Two Dates
Use `date_diff()` and `daysBetween()` to calculate the time between two dates.

Convert Dates to and from Strings
You can convert dates to strings.

Handle Null Dates
Use `is not null` to filter out null dates.

Determine the Day in the Week, Month, Quarter, or Year
These functions return the day of the week, month, quarter, or year, the date of the last day of the week, month, quarter, or year, and the number of days in the quarter or year.

Use Time Zone-Enabled Dates in SAQL Projections

You can project an exact date such as 2017-3-31 23:59:59 or part of a date such as year, month, or day.

Project the entire `CloseDate` field to see the `CloseDate` field for a record. `CloseDate` can be a `DateTime` or `DateOnly` type.

```
q = foreach q generate CloseDate as 'Close Date';
```

Project the year, month, day, and epoch date for a record.

```
q = foreach q generate year('CloseDate') as 'Year', month('CloseDate') as 'Month',
day('CloseDate') as 'Day', epochSecond('CloseDate') as 'Seconds Epoch';
```

Access Date Functions with Time Zone Enabled

Use these functions to get the day, week, year, and other parts of `DateTime` or `DateOnly` fields. The return values are numbers.

- `year (DateTime | DateOnly)`
- `quarter (DateTime | DateOnly)`
- `month (DateTime | DateOnly)`
- `week (DateTime | DateOnly)`
- `day (DateTime | DateOnly)`
- `minute (DateTime | DateOnly)`
- `second (DateTime | DateOnly)`
- `fiscalYear (DateTime | DateOnly)`
- `fiscalQuarter (DateTime | DateOnly)`
- `fiscalMonth (DateTime | DateOnly)`
- `fiscalWeek (DateTime | DateOnly)`
- `epochDay (DateTime | DateOnly)`
- `epochSecond( DateTime | DateOnly)`

Examples

Use `year()`, `month()`, and `day()` to project the year, month, and day for each record. `CloseDate` can be a `DateTime` or `DateOnly` type.

```
q = foreach q generate year('CloseDate') as "Year", month('CloseDate') as "Month",
day('CloseDate') as "Day";
```

Use `month()` to find results that closed in December.

```
q = filter q by month('CloseDate') == 12;
```

Use `month()` to order opportunities by month of close date.

```
q = order q by month('CloseDate');
```

Group By Date

You can group the result of your SAQL query by `DateTime` and `DateOnly` fields.

```
q = group q by 'CloseDate';
```

`CloseDate` can be `DateTime` or `DateOnly`. You can also group by date parts. For example, you can group orders by year and then month.

```
q = group q by year('OrderDate'), month('OrderDate');
```

You can use the `DateTime` or `DateOnly` field to cogroup two datasets. For example, you can group two datasets by year.

```
a = load dataset1;
b = load dataset2;
c = group a by year('CloseDate'), b by year('CloseDate');
e = foreach c generate year(a.'CloseDate') as 'CloseDate A', year(b.'CloseDate') as
'Close Date B', sum(a.Amount) as 'Sum of Amount';
```

Order By Date

You can order the result of your SAQL queries by `DateTime` or `DateOnly`.

Use the date part to order by date before the projection. For example, you can order results by the year that they closed.

```
q = order q by year('CloseDate');
```

To order by date after the projection, use the field you created by projecting a date part. For example, you can order results by the year that they closed.

```
q = foreach q generate year('CloseDate') as "Year Closed";
q = order q by 'Year Closed';
```

Filter By Date

You can filter results by `DateTime` and `DateOnly` fields. Filters can include exact dates, specific date ranges, or relative date ranges.

Type of Filter	Example
exact date range	`q = filter q by year('CloseDate')=='2018';`
specific date range	`q = filter q by year('CloseDate') in [2017..2018];`

relative date range	`q = filter q by CloseDate in ["last 2 years"];`
	`q = filter CloseDate in ["current fiscal_year".."current day"];`
	`q = filter CloseDate in ["2 fiscal_years ago".."current day"];`

Note: Filter with binary comparison operators ==, !=, <, >, <=, and >= only after your query's `foreach` statement, post-projection. For example, include the filter `q = filter q by CloseDate >= "2014-01-01"` post-projection. If you include it pre-projection, the query throws an error. The inability to include filters that use these comparison operators pre-projection is a limitation of the beta release.

You can filter pre- and post-projection with the `IN` comparison operator. The `like` and `matches` operators are not supported for time zone-enabled `DateTime` and `DateOnly` fields.

You can use these relative date keywords:

- `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) ago`

SEE ALSO:

filter

Comparison Operators

Calculate the Time Between Two Dates

Use `date_diff()` and `daysBetween()` to calculate the time between two dates.

Use `now()` to get the current time. You can use these functions with `DateTime`, `DateOnly`, or `Date` types.

date_diff(datepart, startdate, enddate)

Returns an integer representing the interval that has elapsed between two dates.

daysBetween(date1, date2)

Returns the number of days between two dates as an integer.

now()

Returns current datetime in the specified time zone. This function is valid only in a `foreach` statement.

date_diff(*datepart*, *startdate*, *enddate*)

Returns an integer representing the interval that has elapsed between two dates.

datepart	The part of the date to use when calculating the difference. Allowed values are: • `year` • `month` • `quarter` • `day` • `week` • `hour` • `minute` • `second`
startdate	The start date of the interval.
enddate	The end date of the interval.

The difference between two dates is calculated based on the difference in the indicated date parts. For example, the year difference between two dates is calculated by subtracting the year part of ***startdate*** from the year part of ***enddate***.

Suppose `OrderDate` and `ShipDate` are `DateOnly` types. The order date is 31-1-2017 and the ship date is 1-2-2018.

The year difference between the order date and ship date is 1.

```
date_diff("year", 'OrderDate' 'ShipDate');
```

The month difference between the order date and ship date is 2.

```
date_diff("month", 'OrderDate' 'ShipDate');
```

If *startdate* is after ***enddate***, the result is a negative integer.

daysBetween(*date1*, *date2*)

Returns the number of days between two dates as an integer.

For example, display the number of days to close a deal. `OpenDate` and `CloseDate` fields can be `DateTime` or `DateOnly`.

```
q = foreach q generate daysBetween('OpenDate','CloseDate') as "Days to Close";
```

now()

Returns current datetime in the specified time zone. This function is valid only in a `foreach` statement.

Display the number of days an account is opened.

```
q = foreach q generate Account, daysBetween('OrderDate', now()) as "daysOpened";
```

Convert Dates to and from Strings

You can convert dates to strings.

date_to_string(DateTime | DateOnly, formatString)
Converts a date to a string.

toDateTime(epoch)
Converts an epoch to a `DateTime` type.

toDateTime(string, format)
Converts a date in string format to a `DateTime` type. *`format`* specifies the date format and can be any valid date format.

toDateOnly(epoch)
Converts an epoch to a `DateOnly` type.

toDateOnly(string, format)
Converts a date in string format to a `DateOnly` type. *`format`* specifies the date format and can be any valid date format.

`date_to_string(`*`DateTime | DateOnly, formatString`*`)`

Converts a date to a string.

This function takes a `DateTime`, `DateOnly`, or `now()` as its first argument. For the allowed formats, see the Analytics External Data Format Reference.

Use `date_to_string()` to display the close date for your opportunities in the format `yyyy-mm-dd.`

```
q = foreach q generate date_to_string('CloseDate', "yyyy-MM-dd") as "Close Date";
```

`toDateTime(`*`epoch`*`)`

Converts an epoch to a `DateTime` type.

```
q = foreach q generate toDateTime(epoch) as "DateTime";
```

`toDateTime(`*`string, format`*`)`

Converts a date in string format to a `DateTime` type. *`format`* specifies the date format and can be any valid date format.

```
q = foreach q generate toDateTime('CloseDate',"yyyy/MM/dd") as DateTime;
```

`toDateOnly(`*`epoch`*`)`

Converts an epoch to a `DateOnly` type.

```
q = foreach q generate toDateOnly(epoch) as "DateTime";
```

`toDateOnly(string, format)`

Converts a date in string format to a `DateOnly` type. *`format`* specifies the date format and can be any valid date format.

```
q = foreach q generate toDateOnly('CloseDate',"yyyy/MM/dd") as DateTime;
```

Handle Null Dates

Use `is not null` to filter out null dates.

```
q = filter q by 'CloseDate' is not null;
q = foreach q generate 'CloseDate';
```

Projecting null values does not cause an error. For example, this Date Closed field is empty, but no error occurs.

```
q = filter q by year('CloseDate') is null;
q = foreach q generate year('CloseDate') as "Date Closed";
```

Determine the Day in the Week, Month, Quarter, or Year

These functions return the day of the week, month, quarter, or year, the date of the last day of the week, month, quarter, or year, and the number of days in the quarter or year.

day_in_week(date)
Returns an integer representing the day of the week for a specific date. 1 = Sunday, 2 = Monday and so on.

day_in_month(date)
Returns an integer representing the day of the month for a specific date.

day_in_quarter(date)
Returns an integer representing the day of the quarter for a specific date.

day_in_year(date)
Returns an integer representing the day of the quarter for a specific date.

week_last_day(date)
Returns the date of the last day of the week for a specific date.

year_last_day(date)
Returns the date of the last day of the year for a specific date.

quarter_last_day(date)
Returns the date of the last day of the quarter for a specific date.

month_days(date)
Returns the number of days in the month for a specific date.

quarter_days(date)
Returns the number of days in the quarter for a specific date.

year_days(date)
Returns the number of days in the year for a specific date.

day_in_week(*date*)

Returns an integer representing the day of the week for a specific date. 1 = Sunday, 2 = Monday and so on.

```
q = foreach q generate day_in_week('OrderDate') as "Day in Week";
```

day_in_month(*date*)

Returns an integer representing the day of the month for a specific date.

```
q = foreach q generate day_in_month('OrderDate') as "Day in Month";
```

day_in_quarter(*date*)

Returns an integer representing the day of the quarter for a specific date.

```
q = foreach q generate day_in_quarter('OrderDate') as "Day in Quarter";
```

day_in_year(*date*)

Returns an integer representing the day of the quarter for a specific date.

```
q = foreach q generate day_in_quarter('OrderDate') as "Day in Quarter";
```

week_last_day(*date*)

Returns the date of the last day of the week for a specific date.

```
q = foreach q generate week_last_day('BillDate') as "Week Last Day";
```

year_last_day(*date*)

Returns the date of the last day of the year for a specific date.

```
q = foreach q generate year_last_day('BillDate') as "Year Last Day";
```

Note: This function always returns December 31. It's included for uses such as finding the number of days to the year end and for use in a specific locale.

quarter_last_day(*date*)

Returns the date of the last day of the quarter for a specific date.

```
q = foreach q generate quarter_last_day('BillDate') as "Quarter Last Day";
```

month_days(*date*)

Returns the number of days in the month for a specific date.

```
q = foreach q generate month_days(BillDate) as "Days in Billing Month";
```

`quarter_days(date)`

Returns the number of days in the quarter for a specific date.

```
q = foreach q generate quarter_days(BillDate) as "Days in Billing Quarter";
```

`year_days(date)`

Returns the number of days in the year for a specific date.

```
q = foreach q generate year_days(BillDate) as "Days in Billing Year";
```

Work with Custom Fiscal Year Data

After inheriting custom fiscal years, SAQL queries support custom fiscal year data.

Analytics supports custom fiscal year data by generating new fields that describe the custom fiscal year. Each of these new fields is named with the `_Fiscal` suffix. By working with these fields, SAQL supports custom fiscal year data.

Make sure that a dataset's dataflow has run after inheriting custom fiscal years and before writing SAQL based on custom fiscal year data.

Each of the queries in the examples is based off the following dataset. These examples presume that Analytics inherited custom fiscal years that begin on February 1 and end on January 31. Custom fiscal years are defined from 2017 until 2022.

Note: You can't use custom fiscal year data with the fill or timeseries statements.

Opportunity Name	Created Date	Amount
Widgets	2/15/2017	100
Widgets	1/25/2018	200
Widgets	3/30/2018	100
Widgets	3/30/2019	100
Widgets	3/30/2020	100
Widgets	3/30/2021	100
Widgets	3/30/2022	100

Group by a Custom Fiscal Year
Here's how to group by a custom fiscal year.

Filter by a Custom Fiscal Year
Here's how to filter by a custom fiscal year date.

Dates Outside Ranges Defined by Custom Fiscal Year
If your query includes a date that falls outside of a range defined by an inherited fiscal year, SAQL does not return data for that date.

SEE ALSO:

Date Formats and Fiscal Dates for Source Data

Group by a Custom Fiscal Year

Here's how to group by a custom fiscal year.

Example:

```
q = load "opportunities";
q = group q by 'CreatedDate_Year_Fiscal';
q = foreach q generate
   'CreatedDate_Year_Fiscal' as 'Fiscal Year',
   count() as 'count';
q = order q by 'CreatedDate_Year_Fiscal' asc;
q = limit q 2000;
```

The query returns:

Fiscal Year	Count
2017	2
2018	1
2019	1
2020	1
2021	1
2022	1

Filter by a Custom Fiscal Year

Here's how to filter by a custom fiscal year date.

Example:

```
q = load "opportunities";
q = filter q by date('CreatedDate_Year', 'CreatedDate_Month', 'CreatedDate_Day') in
["current fiscal_year".."current_fiscal_year"];
q = group q by 'CreatedDate_Year_Fiscal';
q = foreach q generate
   'CreatedDate_Year_Fiscal' as 'Fiscal Year',
   count() as 'count';
q = order q by 'CreatedDate_Year_Fiscal' asc;
q = limit q 2000;
```

Here's the query output.

Fiscal Year	Count
2020	1

Dates Outside Ranges Defined by Custom Fiscal Year

If your query includes a date that falls outside of a range defined by an inherited fiscal year, SAQL does not return data for that date.

If a date falls outside of a range defined by an inherited custom fiscal year from Salesforce, then SAQL returns `null` for that date. When grouping by a date field that includes dates outside a range defined by an inherited custom fiscal year, no group is returned for undefined dates. If you group data based on non-fiscal periods, dates that aren't included in a custom fiscal year return data as expected.

Example: If your fiscal year ends in March 2021, and a date field, `CreatedDate`, is in April 2021, grouping by `CreatedDate_Month_Fiscal` returns `null` or no group for April 2021. Grouping by `CreatedDate_Month` returns data as expected.

Consider this example dataset.

Opportunity Name	Created Date	Amount
Widgets	2/1/2017	100
Widgets	2/1/2018	100
Widgets	2/1/2019	100
Widgets	2/1/2020	100
Widgets	2/1/2021	100
Widgets	2/1/2022	100
Widgets	2/1/2023	100

In Salesforce, you have custom fiscal years defined as January 1 to December 31 for each year from 2018 through 2022. Inherit them in Analytics by using the **Start Date** setting.

When running a query like this:

```
q = load "opportunities";
q = foreach q generate 'Created_Date' as 'Created Date', Created_Date_Year_Fiscal as 'Fiscal
 Year';
q = limit q 2000;
```

SAQL returns these results:

Created Date	Fiscal Year
2/1/2017	-
2/1/2018	2018
2/1/2019	2019
2/1/2020	2020
2/1/2021	2021
2/1/2022	2022
2/1/2023	-

Because a custom fiscal year definition doesn't include 2/1/2017 or 2/1/2023, SAQL returns `null`.

Now, let's group the dataset.

```
q = load "opportunities";
q = group q by 'Created_Date_Year_Fiscal';
q = foreach q generate 'Created Date_Year_Fiscal' as 'Fiscal Year', count() as 'Count';
q = order q by 'Created_Date_Year_Fiscal';
q = limit q 2000;
```

SAQL returns these results:

Fiscal Year	Count
2018	1
2019	1
2020	1
2021	1
2022	1

Since the custom fiscal year definition doesn't include 2/1/2017 or 2/1/2023, the query excludes these dates from the results.

String Functions

Use SAQL string functions to format your measure and dimension fields.

ascii()
Returns the UTF-8 code value of a character *n*.

chr()
Returns the UTF-8 character of integer *n*.

ends_with()
Returns `true` if the string ends with the specified characters.

index_of()
Returns the location (index) of the specified characters.

len()
Returns the number of characters in the string.

lower()
Returns a copy of the string with all characters in lower case.

ltrim()
Removes the specified characters from the beginning of a string.

mv_to_string()
Converts multivalue fields to string fields.

number_to_string
Converts a number literal to a string literal.

replace()
Replaces a substring with the specified characters.

rtrim()
Removes the specified characters from the end of a string.

starts_with()
Returns `true` if the string starts with the specified characters.

string_to_number
Converts a string literal to a number literal.

substr()
Returns a substring that starts at the specified position. You can also specify the length of the substring to return.

trim()
Removes the specified substring from the beginning and the end of a string.

upper()
Returns a copy of the string with all characters in upper case.

ascii()

Returns the UTF-8 code value of a character *n*.

Syntax

`ascii(`*n*`)`

Usage

Returns `null` if *n* is `null`. The `null` character (0) is not allowed.

Example

```
q = foreach q generate ascii("a") as int_value;
- -int_value == 97
```

chr()

Returns the UTF-8 character of integer *n*.

Syntax

`chr(`*n*`)`

Usage

Returns `null` if *n* is `null`.

Example

```
q = foreach q generate chr(97) as char_value;
- -char_value == a
```

ends_with()

Returns `true` if the string ends with the specified characters.

Syntax

`ends_with(`***`string, suffix`***`)`

Usage

Returns `true` if ends with *`suffix`*, otherwise returns `false`. 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`.

Example

```
ends_with("FIT", "T") == true
ends_with("FIT", "BIT") == false
```

index_of()

Returns the location (index) of the specified characters.

Syntax

`index_of(`*`string`*, *`searchStr`* [,*`position`* [, *`occurence`*]])

Usage

This function returns the index 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.

`occurrence` is an optional integer, with a default value of `1` . You can use this parameter to specify which occurrence of *`searchStr`* to search for. For example, if there is more than one occurrence of *`searchStr`*, 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`*.

Example

```
-- return the first occurrence of "a", starting at the beginning.
-- The result is 2.
```

```
q = foreach q generate index_of("Hawaii", "a") as 'Index';

-- return the second occurrence of "a", starting at the beginning
-- the result is 4
q = foreach q generate index_of("Hawaii", "a",1, 2) as 'Index';


-- return the first occurrence of "a", starting at the third position
-- the result is 4
q = foreach q generate index_of("Hawaii", "a",3) as 'Index';
```

`len()`

Returns the number of characters in the string.

Syntax

`len(`*`string`*`)`

Usage

Leading and trailing whitespace characters are included in the length returned. Returns `null` if *`string`* is `null`.

Example

```
len("starfox") == 7
len(" rocket ") == 8
len("□") == 1
len("") == 0
```

`lower()`

Returns a copy of the string with all characters in lower case.

Syntax

`lower(`*`string`*`)`

Usage

Returns `null` if *`string`* is `null`.

Example

```
lower("JAVA") == "java"
```

`ltrim()`

Removes the specified characters from the beginning of a string.

Syntax

`ltrim(`*`string`*`,`*`substr`*`)`

Usage

Removes every instance of each character in *`substr`* from the beginning of *`string`*. This function is case-sensitive. To remove leading spaces, do not specify a value for *`substr`*.

Example

This example shows that `ltrim` removes the specified characters from the beginning of a string. This function is case-sensitive.

```
q = load "test";
q = foreach q generate 'Company' as 'Company', ltrim('Company',"abc") as 'ltrim abc',
    ltrim('Company',"cba") as 'ltrim cba', ltrim('Company',"ab") as 'ltrim ab',
    ltrim('Company',"bc") as 'ltrim bc';
```

Company	ltrim abc	ltrim cba	ltrim ab	lt
CompanyABCABC	CompanyABCABC	CompanyABCABC	CompanyABCABC	C
abcabcCompany	Company	Company	cabcCompany	a
ABCABCCompany	ABCABCCompany	ABCABCCompany	ABCABCCompany	A

mv_to_string()

Converts multivalue fields to string fields.

Syntax

`mv_to_string(`*`multivalue_column_name`*`, `*`delimeter`*`)`

`multivalue_column_name`
Name of the multivalue field to be converted to a string.

`delimiter`
Optional. The characters used to delimit values in the converted string. Maximum length is 2 characters.

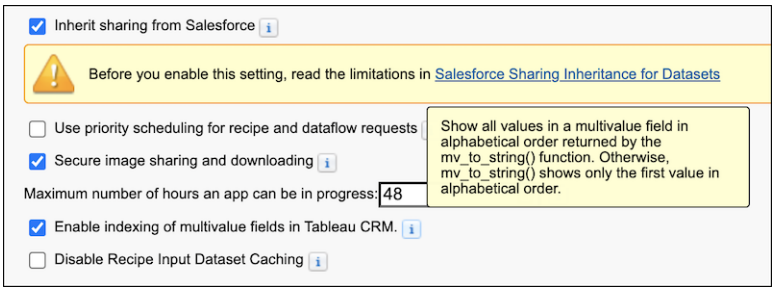
Usage

Returns an alphabetically-sorted, delimited string representation of a multivalue field. The default delimiter is a comma followed by a space (,).

`mv_to_string()` applies to non-grouped streams only. You can run filtering or grouping on a multivalue field post-projection.

Note: To enable multivalue fields, you must select the **Enable indexing of multivalue fields in Analytics** preference in Setup. If you run `mv_to_string()` without the preference selected, the function returns the first value in the first field only.

1. From Setup, enter *`Analytics`* in the Quick Find box.
2. Select **Settings** from the list of Analytics options.
3. In Settings, click the checkbox for **Enable indexing of multivalue fields in CRM Analytics**.



Example

This query returns values of the `Accounts Team` as a string delimited by a comma and space, in alphabetical order.

```
q = load "Accounts";
q = foreach q generate
    'Account' as 'Account';
    mv_to_string('Account_Team') as 'Account Team';
```

Account	Account Team
Acme	Fred Williamson, Hank Chen, Sarah Vasquez
DTC Electronics	Brian Alison, Tessa McNaley
Salesforce	Nadia Smith

Example

This query returns the values of `Accounts Team` as a string delimited by two semicolons (;;) in alphabetical order.

```
q = load "Accounts";
q = foreach q generate
    'Account' as 'Account';
    mv_to_string('Account_Team', ";;") as 'Account Team';
```

Account	Account Team
Acme	Fred Williamson;;Hank Chen;;Sarah Vasquez
DTC Electronics	Brian Alison;;Tessa McNaley

Account	Account Team
Salesforce	Nadia Smith

SEE ALSO:

Multivalue Field

number_to_string

Converts a number literal to a string literal.

Syntax

`number_to_string(number, number_format)`

Usage

Returns the string representation of *number*. Use *number_format* to specify the format of the string, for example as currency or with two decimal places. *number_format* can specify seperate formats for positive and negative numbers:

- `number_to_string(number, number_format)`

  The format specified by *number_format* is used for both positive and negative numbers.

- `number_to_string(number, <POSITIVE>;<NEGATIVE>)`

  If *number* is positive, the number format specified by *<POSITIVE>* is used. If *number* is negative, the number format specified by *<NEGATIVE>* is used. Note the semicolon separating the two specified formats.

You can specify the format with these characters:

- 0, #, decimal point (.)
- Thousands separator (,)
- Percentage (%)
- Leading and trailing characters: $, +, -, (,), :, !, ^,&,',~,{,}

Example

Display the number amount as a string, formatted as currency:

```
q = foreach q generate 'Amount' as 'Amount', number_to_string('Amount',"$#,###.00") as
'NumberAmount';
```

Amount	NumberAmount
397,280	$397,280.00

Example

Suppose that you have a measure field with the format shown in **Number You Start With**. Use the format shown in **number_format** to display this number as a shown in **Resulting String**.

Initial Number	number_format	Resulting String
1234.56	####.#	1234.6
8.9	#.000	8.900
.631	0.#	0.6
12	#.0#	12.0
1234.568	#.0#	1234.57
12000	#,###	12,000
12000	#,	12
12200000	0.0,,	12.2
12	00000	00012
0.03457	#.00%	3.46%
12.3	$#.00;($#.00)	$12.30
-12.3	$#.00;($#.00)	($12.30)
32	+;-	+
-32	+;-	-

`replace()`

Replaces a substring with the specified characters.

Syntax

`replace(`*`string, searchStr, replaceStr`*`)`

Usage

This function replaces *`searchStr`* with *`replaceStr`*, then returns the modified string. If any of the parameters are `null`, then the function returns `null`. If *`searchStr`* is an empty string, the function returns `null`. This function is case-sensitive.

Example

```
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()`

Removes the specified characters from the end of a string.

Syntax

`rtrim(string,substr)`

Usage

Removes every instance of each character in *substr* from the end of *string*. This function is case-sensitive. To remove trailing spaces, do not specify a value for *substr*.

Example

This example shows that `rtrim` removes the specified characters from the end of a string. This function is case-sensitive.

```
q = load "test";
q = foreach q generate 'Company' as 'Company', rtrim('Company',"abc") as 'rtrim abc',
    rtrim('Company',"cba") as 'rtrim cba', rtrim('Company',"ab") as 'rtrim ab',
    rtrim('Company',"ac") as 'rtrim ac';
```

Company	rtrim abc	rtrim cba	rtrim ab	rt
Companyabcabc	Company	Company	Companyabcabc	C
CompanyABCABC	CompanyABCABC	CompanyABCABC	CompanyABCABC	C

starts_with()

Returns `true` if the string starts with the specified characters.

Syntax

`starts_with(`***string, prefix***`)`

Usage

Returns `true` if *string* starts with *prefix*, otherwise returns `false`. 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`.

Example

Suppose that you want to count the opportunities where the owner role starts with "Sales". Use `starts_with()` in a `case` statement.

```
q = load "DTC_Opportunity";

-- Select rows where the owner roles starts with "Sales"
q = foreach q generate count() as 'count', (case
 when starts_with('Owner_Role', "Sales") then 'Owner_Role'
end) as 'Owner_Role';
```

```
q = group q by 'Owner_Role';
q = foreach q generate count() as 'count', 'Owner_Role' as 'Owner_Role';
```

The resulting chart shows the number of opportunities where the owner role starts with "Sales", grouped by owner role.

Owner Role	Count of Rows
Sales AMER	27
Sales EMEA	28
Sales WW	45

string_to_number

Converts a string literal to a number literal.

Syntax

`string_to_number(`***string***`)`

Usage

If the string can't be parsed as a number, the query fails.

Example

```
-- creates a field called "Number" that contains the number 12345

q = foreach q generate string_to_number("12345") as 'Number';
```

substr()

Returns a substring that starts at the specified position. You can also specify the length of the substring to return.

Syntax

`substr(`***string*****,*****position***`[, `***length***`])`

Usage

`substr` returns the characters in *`string`*, starting at position *`position`*. If you specify *`length`*, this function returns *`length`* number of characters. If any of the parameters are *`null`*, then the function returns *`null`*. *`length`* is optional.

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.

Example

```
-- we want a substring that is one character long, starting at position 1.
-- The character "C" is returned.
substr("CRM", 1, 1)

-- we want a substring that is 2 characters long, starting at position 1
-- The string "CR" is returned
substr("CRM", 1, 2) == "CR"

-- we want a substring that is two characters long, starting from the *end* of the string
-- The string "RM" is returned
substr("CRM", -2, 2) == "RM"

-- we want to get the first 10 characters from this string
-- the string "2018-03-16" is returned
substr("2018-03-16T00:00:03.000Z",10)
```

Example

Suppose that you want to display the current time, but not the current date. Use `substr()` to return the last 11 characters from `date_to_string()` .

```
q = foreach q generate substr(date_to_string(now(), "yyyy-MM-dd HH:mm:ss"), 11) as 'Time
Now';
```

trim()

Removes the specified substring from the beginning and the end of a string.

Syntax

trim(*string*,*substr*)

Usage

This function removes *substr* from the beginning and end of *string*, then returns the result. To remove leading and trailing spaces, do not specify a value for *substr*.

Example

```
--  the resulting string in both cases is 'MyString';
q = foreach q generate trim("abcMyStringabc","abc") as 'Trimmed String';
q = foreach q generate trim("     MyString     ") as 'Trimmed String';
```

`upper()`

Returns a copy of the string with all characters in upper case.

Syntax

`upper(`*`string`*`)`

Usage

Returns `null` if *`string`* is `null`.

Example

```
upper("java") == "JAVA"
```

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.

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 function can only be used in a `foreach` statement.

acos(n)
Returns the arccosine value of radians value *`n`*. *`n`* can be any real numeric value in the range of -1 <= *`n`* <= 1. If `null` is passed as an argument, `acos()` returns `null`. This function can only be used in a `foreach` statement.

asin(n)
Returns the arcsine value of radians value *`n`*. *`n`* can be any real numeric value in the range of -1 <= *`n`* <= 1. If `null` is passed as an argument, `asin()` returns `null`.This function can only be used in a `foreach` statement.

atan(n)
Returns the arctangent value of radians value *`n`*. *`n`* can be any real numeric value in the range of -1e308 <= *`n`* <= 1e308. If `null` is passed as an argument, `atan()` returns `null`. This function can only be used in a `foreach` statement.

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 function can only be used in a `foreach` statement.

cos(n)
Returns the cosine value of radians value *`n`*. *`n`* can be any real numeric value in the range of -1e308 <= *`n`* <= 1e308. If `null` is passed as an argument, `cos()` returns `null`. This function can only be used in a `foreach` statement.

degrees(n)
Returns the degrees value of a radians value *`n`*. *`n`* can be any real numeric value in the range of -1e308 <= *`n`* <= 1e308. If `null` is passed as an argument, `degrees()` returns `null`. This function can only be used in a `foreach` statement.

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 doesn't result in 0 is 3e-324. `n` can be any real numeric value in the range of -1e308 <= n <= 700. This function can only be used in a `foreach` statement.

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 function can only be used in a `foreach` statement.

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 value for `m` or `n` that will not produce 0 is log(10, 0.3e-323). This function can only be used in a `foreach` statement.

pi()

Returns the value of π, where π=3.14139265. This function can only be used in a `foreach` statement.

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. This function can only be used in a `foreach` statement.

radians(n)

Returns the radians value of a degrees value `n`. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308. If `null` is passed as an argument, `radians()` returns `null`. This function can only be used in a `foreach` statement.

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 function can only be used in a `foreach` statement.

sign(n)

Returns 1 if the numeric value, `n` is positive. It returns -1 if the `n` is negative, and 0 if `n` is 0. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308. If `null` is passed as an argument, `sign()` returns `null`. This function can only be used in a `foreach` statement.

sin(n)

Returns the sine value of radians value `n`. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308. If `null` is passed as an argument, `sin()` returns `null`. This function can only be used in a `foreach` statement.

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. This function can only be used in a `foreach` statement.

tan(n)

Returns the tangent value of radians value `n`. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308. If `null` is passed as an argument, `tan()` returns `null`. This function can only be used in a `foreach` statement.

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 function can only be used in a `foreach` statement.

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 function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate abs(pct_change) as pct_magnitude;
```

acos(*n*)

Returns the arccosine value of radians value *n*. *n* can be any real numeric value in the range of -1 <= *n* <= 1. If `null` is passed as an argument, `acos()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate acos(radians) as arccosine;
```

asin(*n*)

Returns the arcsine value of radians value *n*. *n* can be any real numeric value in the range of -1 <= *n* <= 1. If `null` is passed as an argument, `asin()` returns `null`.This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate asin(radians) as arcsine;
```

atan(*n*)

Returns the arctangent value of radians value *n*. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `atan()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate atan(radians) as arctangent;
```

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 function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate ceil(miles) as distance;
```

cos(*n*)

Returns the cosine value of radians value *n*. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `cos()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate cos(radians) as cosine;
```

degrees(n)

Returns the degrees value of a radians value `n`. `n` can be any real numeric value in the range of -1e308 <= `n` <= 1e308. If `null` is passed as an argument, `degrees()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate degrees(radians) as degrees;
```

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 doesn't result in 0 is 3e-324. `n` can be any real numeric value in the range of -1e308 <= n <= 700. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate exp(value) as value;
q = filter q by exp(value) < 5;
```

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 function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate floor(miles) as distance;
```

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 value for `m` or `n` that will not produce 0 is log(10, 0.3e-323). This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate log(10, Population) as Population;
q = filter q by log(10, Population) < 15;
```

pi()

Returns the value of π, where π=3.14139265. This function can only be used in a `foreach` statement.

Example:

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

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. This function can only be used in a `foreach` statement.

- 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.

Example:

```
q = foreach q generate power(length, 2) as area, length;
q = filter q by power(length, 2) > 10;
```

radians(*n*)

Returns the radians value of a degrees value *n*. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `radians()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate radians(degrees) as radians;
```

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 function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate round(Price, 2) as Price;
```

sign(*n*)

Returns 1 if the numeric value, *n* is positive. It returns -1 if the *n* is negative, and 0 if *n* is 0. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `sign()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate sign(value) as value;
```

sin(*n*)

Returns the sine value of radians value *n*. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `sin()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate sin(radians) as sine;
```

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. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate sqrt(value) as value;
q = filter q by sqrt(value) < 10;
```

tan(*n*)

Returns the tangent value of radians value *n*. *n* can be any real numeric value in the range of -1e308 <= *n* <= 1e308. If `null` is passed as an argument, `tan()` returns `null`. This function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate tan(radians) as tangent;
```

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 function can only be used in a `foreach` statement.

Example:

```
q = foreach q generate trunc(Price, 2) as Price;
```

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.

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.

Example - Dynamically Display Your Top Five Reps

Use windowing to create a chart that dynamically displays your top-five reps for each country. The chart updates continuously as opportunities are won. The example uses windowing to calculate:

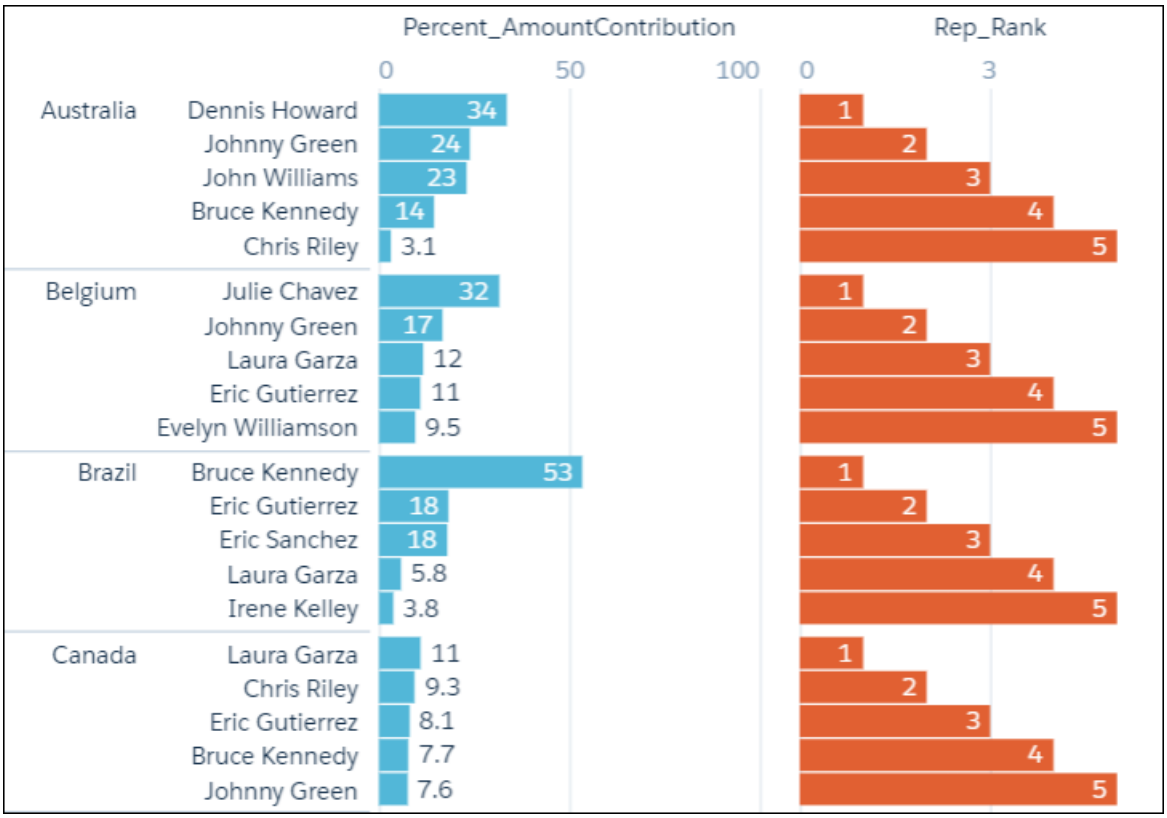
- Percentage contribution that each rep made to the total amount, partitioned by country
- Ranking of the rep's contribution, partitioned by country

These calculations let us display the top-five reps in each country.

```
q = load "DTC_Opportunity_SAMPLE";
q = group q by ('Billing_Country', 'Account_Owner');

q = foreach q generate 'Billing_Country', 'Account_Owner',

-- sum(Amount) is the total amount for a single rep in the current country
-- sum(sum('Amount') is the total amount for ALL reps in the current country
-- sum(Amount) / sum(sum('Amount') calculates the percentage that each rep contributed
-- to the total amount in the current country
((sum('Amount')/sum(sum('Amount'))
```

```
-- [..] means "include all records in the partition"
-- "by Billing_Country" means partition, or group, by country
over ([..] partition by 'Billing_Country')) * 100) as 'Percent_AmountContribution',

-- rank the percent contribution and partition by the country
rank() over ([..] partition by ('Billing_Country') order by sum('Amount') desc ) as
'Rep_Rank';

-- filter to include only the top 5 reps
q = filter q by 'Rep_Rank' <=5;
```

The resulting graph shows the top-five reps in each country and displays each rep's ranking.



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
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

Year	Quarter	sum_amt	r_sum
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%**

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

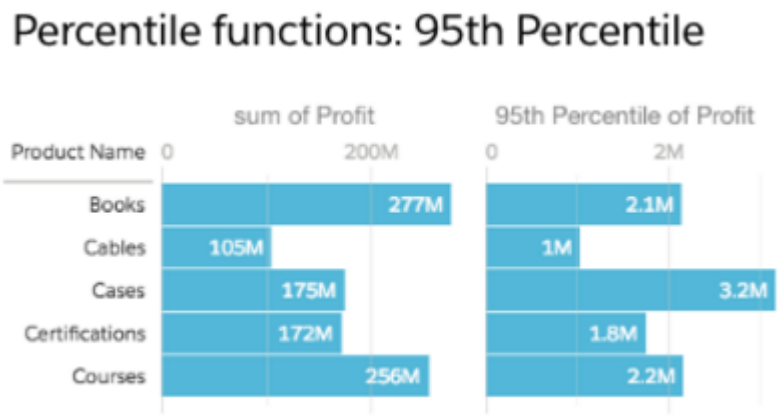
This query shows the top 3 performing quarters in a year.

```
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
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.

SEE ALSO:

- Windowing Functions
- Windowing Functions

coalesce

Use `coalesce()` to get the first non-null value from a list of parameters, or to replace nulls with a different value.

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

Example: Left Outer Cogroup with `coalesce()`

A left outer cogroup combines the right data stream with the left data stream. If a record on the left stream does not have a match on the right stream, the missing right value comes through as null. To replace null values with a different value, use `coalesce()`.

For example, suppose that you have a dataset of meeting information from the Salesforce Event object, and you join it with data from the Salesforce Opportunity object. This shows amount won with the total time spent in meetings.

```
ops = load "Ops";
meetings = load "Meetings";
q = cogroup ops by 'Account' left, meetings  by 'Company' ;
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
sum(meetings.'MeetingDuration') as 'TimeSpent';
```

It looks like we had no meetings with Zen Retreats.

Account	Sum of Amount	TimeSpent
FreshMeals	3.4	4
Shoes2Go	4.5	7
ZenRetreats	2	-
ZipBikeShare	1.1	4

Let's use `coalesce()` to change that null value to a zero.

```
ops = load "Ops";
meetings = load "Meetings";
q = cogroup ops by 'Account' left, meetings  by 'Company' ;

--use coalesce() to replace null values with zero
q = foreach q generate ops.'Account' as 'Account', sum(ops.'Amount') as 'sum_Amount',
coalesce(sum(meetings.'MeetingDuration'), 0) as 'TimeSpent';
```

Account	Sum of Amount	TimeSpent
FreshMeals	3.4	4
Shoes2Go	4.5	7
ZenRetreats	2	0
ZipBikeShare	1.1	4

QUERY PERFORMANCE

Here are some guidelines for structuring your queries to improve performance.

Speed Up Queries with Dataflow Transformations

To speed up your queries and reduce the number of network round trips, perform data transformations in the ELT process instead of in the query.

Example: `GEO` Field

Let's say you have a dataset with the `GEO` field that contains the value `JP`, and you want to replace this value with `Japan`. One solution is to add a `case` statement to your query.

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

Running this query on each row in a dataset is time-consuming. A faster approach is to add the `case` statement to the `saqlExpression` field in the dataflow's `computeExpression` transformation. Moving the `case` statement from the query to the ELT process reduces the query's network round trips.

```
"parameters": {
    "source": "Opportunity_Data",
```

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

Tip: You can also improve query performance by shortening decimal values in your dataflow. For example, if the numbers in your dataset have a single decimal digit, such as 9.1 or 924.3, set `scale` to 1 rather than 4 in the `computeExpression` transformation. Restricting the decimal value only impacts storage only. SAQL performs query calculations with all decimal values intact.

Example: Date Format

To change the date format, you can add an intermediate query to filter the stream based on the list selector values. Adding an extra filter creates another network trip. Instead, transform the values in the `computeExpression` transformation, which you can use with SAQL date functions.

```
"parameters":{
    "source":"Opportunity_Data",
    "mergeWithSource":true,
    "computedFields":[ {
        "name":"UIFormattedDate",
        "type": "Text",
        "saqlExpression":"date_to_string(toDate(Date_sec_epoch), "yyyy-MM-dd")" } ] }}
```

SEE ALSO:

Simple case Operator

computeExpression Transformation

Limit Multivalue Fields

Multivalue fields can cause poor performance. The behavior of these fields is undefined for `group-by` and `foreach` statements. If possible, write your query so that the fields are referenced only in filters.

Note: To work with multivalue fields, from Setup, in the Quick Find box, enter *Analytics*, and then select **Settings**. In Settings, click the checkbox for **Enable indexing of multivalue fields in CRM Analytics**. If you don't select this preference, the `mv_to_string()` function returns only the first value in the field. See mv_to_string() on page 107 for more information.

Even with indexing enabled, multivalue fields in multilevel grouping, such as `group by(Year, Region)`, can cause poor performance.

Here's FlightsMV, a sample dataset of flight information.

airline	flight_num	origin	dest	pilot	airplane	flight_classes	flight_attendants	distance	num_passengers
southwest	sw301	lax	sfo	john	boeing 737-700	business;economy	mary;sally;mark	1000	100

airline	flight_num	origin	dest	pilot	airplane	flight_classes	flight_attendants	distance	num_passengers
united	u321	lax	sfo	mark	boeing 737-900	first;business;economy	jen;sophia;emma;kate	1000	200
alaska	as400	lax	sfo	tim	airbus A320	business;economy	mark;leila;brad	1000	100
delta	d301	lax	sfo	martin	airbus A321	business;economy	sarah;maria	1000	100
southwest	sw302	sfo	lax	-		business;economy	-	1000	100
united	u322	sfo	lax	john	boeing 737-700	first;business;economy	sarah;martin	1000	200
alaska	as401	sfo	lax	mark	boeing 737-900	business;economy	maria;kate;sarah	1000	100
delta	d302	sfo	lax	tim	airbus A320	business;economy	emma;jen;sophia;kate	1000	100
southwest	sw303	lax	jfk	robert	airbus A321	business;economy	leila;mark;brad	1000	100
united	u323	lax	jfk	maria		first;business;economy	-	1000	200
alaska	as403	lax	jfk	mark	boeing 737-700	business;economy	-	1000	100
delta	d303	lax	jfk	tim	boeing 737-900	business;economy	maria;sarah	1000	100
southwest	sw304	jfk	lax	robert	airbus A320	business;economy	martin;sarah	1000	100
united	u324	jfk	lax	-	airbus A321	first;business;economy	kate;kate;sarah	1000	200
alaska	as404	jfk	lax	john		business;economy	sophia;emma;kate	1000	100
delta	d304	jfk	lax	mark	boeing 737-700	business;economy	-	1000	100
southwest	sw303	lax	ord	martin	boeing 737-900	business;economy	-	1000	100
united	u323	lax	ord	robert	airbus A320	first;business;economy		1000	200
alaska	as403	lax	ord	-	airbus A321	business;economy	brad;mark;leila	1000	100
delta	d303	lax	ord	john	-	business;economy	sarah;maria	1000	100

The `flight_attendants` column contains multivalue fields. Let's write a query to filter on the rows where `maria` is listed as a flight attendant.

```
q = load "FlightsMV";
q = filter q by 'flight_attendants'=="maria";
q = foreach q generate 'airplane' as 'airplane', 'distance' as 'distance',
'flight_attendants' as 'flight_attendants', 'flight_num' as 'flight_num', 'id' as 'id',
'num_passengers' as 'num_passengers', 'origin' as 'origin', 'pilot' as 'pilot';
```

airplane	distance	flight_attendants	flight_num	id	num_passengers	origin	pilot
boeing 737-700	1000	kate	sw301	1	100	lax	john
airbus A321	1000	maria	d301	4	100	lax	martin
boeing 737-900	1000	kate	as401	7	100	sfo	mark
boeing 737-900	1000	maria	d303	12	100	lax	tim
airbus A321	1000	kate	u324	14	200	jfk	-
-	1000	maria	d303	20	100	lax	john

The results display the rows that include `maria`. The `flight_attendants` field displays only one flight attendant name when the field is multivalue. To return all the names, use the `mv_to_string()` function.

```
q = load "FlightsMV";
q = filter q by 'flight_attendants'=="maria";
q = foreach q generate 'airplane' as 'airplane', 'distance' as 'distance',
mv_to_string('flight_attendants') as 'flight_attendants', 'flight_num' as 'flight_num',
'id' as 'id', 'num_passengers' as 'num_passengers', 'origin' as 'origin', 'pilot' as
'pilot';
```

SEE ALSO:

Multivalue Field

Use Group and Filter Pre-projection

Improve query performance by moving group and filter operations on simple fields before the `foreach` statement. A simple field is projected as-is and doesn't have additional expressions.

Projection refers to the subset of columns that your query returns. In SAQL, projection occurs in the `foreach` statement, where the query performs an operation on each row in the dataset.

Note: SAQL supports only pre-projection filters that follow this format: ***`fieldName operatorName constant`***. For example, you can include `q = filter q by Category;`, but not `q = filter q by Discount > 1;`. The same applies to grouping. For example, you can include `q = group q by Category;`, but not `q = group q by Discount > 1;`.

Example: Filter

In this query, the `filter` statement occurs post-projection.

```
q = load "Superstore";
q = foreach q generate 'Category' as 'Store_Category', 'Sub_Category' as
'Store_Sub_Category';
q = filter q by 'Store_Category'=="Furniture";
```

Here, we move the filter to pre-projection. Because the `Category` field occurs before the `foreach` statement, it doesn't have an alias.

```
q = load "Superstore";
q = filter q by 'Category'=="Furniture";
q = foreach q generate 'Category' as 'Store_Category', 'Sub_Category' as
'Store_Sub_Category';
```

Example: Group

In this example, the query first creates two new fields: `Detailed_Category`, a combination of the `Category` and `Sub_Category` fields, and `Adjusted_Discount`. After grouping the results by `Detailed_Category`, the second `foreach` statement takes the average of `Adjusted_Discount` for each `Detailed_Category`.

```
q = load "Superstore";
q = foreach q generate 'Category'+ "-" + 'Sub_Category' as 'Detailed_Category', 2*'Discount'
 as 'Adjusted_Discount';
q = group q by 'Detailed_Category';
q = foreach q generate 'Detailed_Category', avg('Adjusted_Discount') as
'Avg_Adjusted_Discount';
```

Instead of using two `foreach` statements, group by `Sub_Category` pre-projection, and add its alias in the `foreach` statement.

```
q = load "Superstore";
q = group q by ('Category', 'Sub_Category');
q = foreach q generate 'Category'+ "-" + 'Sub_Category' as 'Detailed_Category',
2*avg('Discount') as  'Avg_Adjusted_Discount';
```

SEE ALSO:

- group-by
- filter

Remove Redundant Projections

To improve memory usage and performance costs, remove unnecessary projections from your queries and load only the data required. If you have to perform an operation, include pre-projection statements as needed.

Here's an example of a query with an unnecessary projection.

```
q = load "Superstore";
q = foreach q generate 'Category';
q = group q by 'Category';
q = foreach q generate 'Category', count() as 'count';
```

The first `foreach` statement projects the `Category` field, which is already included in the dataset. Since we're not performing any operation on the field, we can remove it.

```
q = load "Superstore";
q = group q by 'Category';
q = foreach q generate 'Category', count() as 'count';
```

Here's an example with an implicit `cogroup`.

```
a = load "Customer_Data";
a = foreach a generate 'Customer_Name';
b = load "Superstore";
b = foreach b generate 'Customer_Name';
a = group a by 'Customer_Name' full, b by 'Customer_Name';
a = foreach a generate coalesce(a.'Customer_Name', b.'Customer_Name') as 'Customer_Name',
 count('a') as 'Superstore', count('b') as 'Customer_data';
```

In this example, the `foreach` statements that follow loading the "Customer_Data" and the "Superstore" datasets are unnecessary, since they're projecting the `Customer_Name` fields without any additional action. You can group the fields pre-projection.

```
a = load "Customer_Data";
b = load "Superstore";
a = group a by 'Customer_Name' full, b by 'Customer_Name';
a = foreach a generate coalesce(a.'Customer_Name', b.'Customer_Name') as 'Customer_Name',
 count('a') as 'Superstore', count('b') as 'Customer_data';
```

Check for Redundant Filters

Binding and faceting in your dashboard design can generate redundant filters. Check the SAQL queries that your dashboard produces to remove unnecessary filters. Consider how a filter in your dashboard UI interacts with one in your query and vice versa.

Limit the Use of `unique()`

`unique()` can affect query performance for large datasets that have over 100 million rows and include more than one million unique values. For large datasets, `unique()` is faster for measures than for dimensions. If counting the number of unique string values causes performance issues, convert the string to a number. For example, use a hash of the string value or refer to the index of the string in a sorted list of string values.

Note: Counting unique values can impact performance, but counting the total number of rows in a dataset doesn't.