

## Salesforce Wave Best Practices Series

# Building Apps Using Custom SAQL

## *Implementation Best Practices*



*Justin Kim, Data Scientist, ASG*  
*Chocks Kandasamy, Architect, ASG*  
*Terrence Tse, Architect, ASG*  
*Terence Wilson, Senior Architect, ASG*  
*Kaushik Ruparel, Director, ASG*

© Copyright 2000–2016 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

# CONTENTS

## [Backdrop](#)

### [Dataset](#)

### [Anatomy of a Wave Dashboard](#)

#### [Metadata](#)

#### [State](#)

#### [Steps](#)

#### [Widgets](#)

#### [Layouts](#)

### [Wave Runtime](#)

## [SAQL Basics](#)

### [Apache Pig-Latin](#)

### [Salesforce Analytics Query Language \(SAQL\)](#)

### [Elements of a SAQL Query](#)

### [Windowing](#)

### [Use Cases](#)

### [Pros and Cons](#)

## [SAQL Examples](#)

### [Derived Measures](#)

### [Derived Dimensions](#)

## [Bindings](#)

### [Binding Operations](#)

### [Results Binding](#)

#### [Null handling in Results Binding](#)

#### [Null Handling Alternative without Bindings](#)

#### [Using Intermediate steps in Results Binding](#)

### [Selection Binding](#)

#### [Dynamic Sort Order](#)

#### [Dynamic Row Limits](#)

#### [Dynamic Filter \(String Dimension\)](#)

#### [Dynamic Filter \(Full Date\)](#)

#### [Dynamic Filter \(Partial Date\)](#)

#### [Dynamic Filter \(Relative Date\)](#)

#### [Dynamic Grouping](#)

#### [Dynamic Measure](#)

## [Binding Summary](#)

## [Field-to-Field Filters](#)

### [Dimensions](#)

### [Dates](#)

### [Measures](#)

## [Calculations across multiple data streams \(datasets\)](#)

[Joins vs. Co-group](#)

[Sample Data](#)

[Opportunity](#)

[Quota](#)

[Inner Co-Group](#)

[Left Outer Co-Group with Coalesce](#)

[Right Outer Co-Group with Coalesce](#)

[Full Outer Co-Group with Coalesce](#)

[Co-Group as Filter](#)

[Union](#)

[Case Statement](#)

[Binning](#)

[Threshold](#)

[Time-based analysis](#)

[MTD \(Month To Date\)](#)

[YTD \(Year To Date\)](#)

[QTD \(Quarter To Date\)](#)

[QoQ \(Quarter over Quarter\)](#)

[Windowing Alternative](#)

[QTD over QTD](#)

[Current QTD over last year same QTD](#)

[Most recent available month](#)

[MoM on most recent available month](#)

[Windowing Alternative](#)

[Timeline Comparisons](#)

[Advanced Math Operations](#)

[Correlation Coefficient](#)

[Simple Linear Regression](#)

[Standard Score Normalization](#)

[Mean and Standard Deviation](#)

[Quartiles](#)

[Percentiles](#)

[Windowing Functions](#)

[14 day moving average](#)

[Percent of year](#)

[Pareto](#)

[Top 5 Reps in a Region](#)

[Sales Wave Pipeline Trending Customizations](#)

[Date Range](#)

[Filters](#)

[Relabeling Buckets](#)

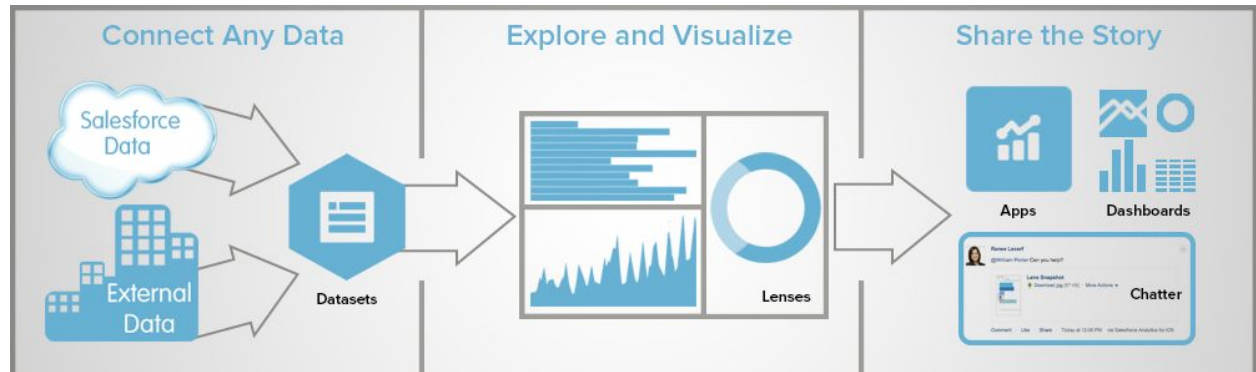
[Other Customizations](#)

[What's New](#)

[Version 2.0](#)[Version 2.1](#)[Credits](#)

# Backdrop

Salesforce Analytics Cloud is a cloud-based platform for connecting data from multiple sources, creating interactive views of that data, and sharing those views in dashboards. It's a better way to distribute insight to business users so they can understand and take action on changing information.

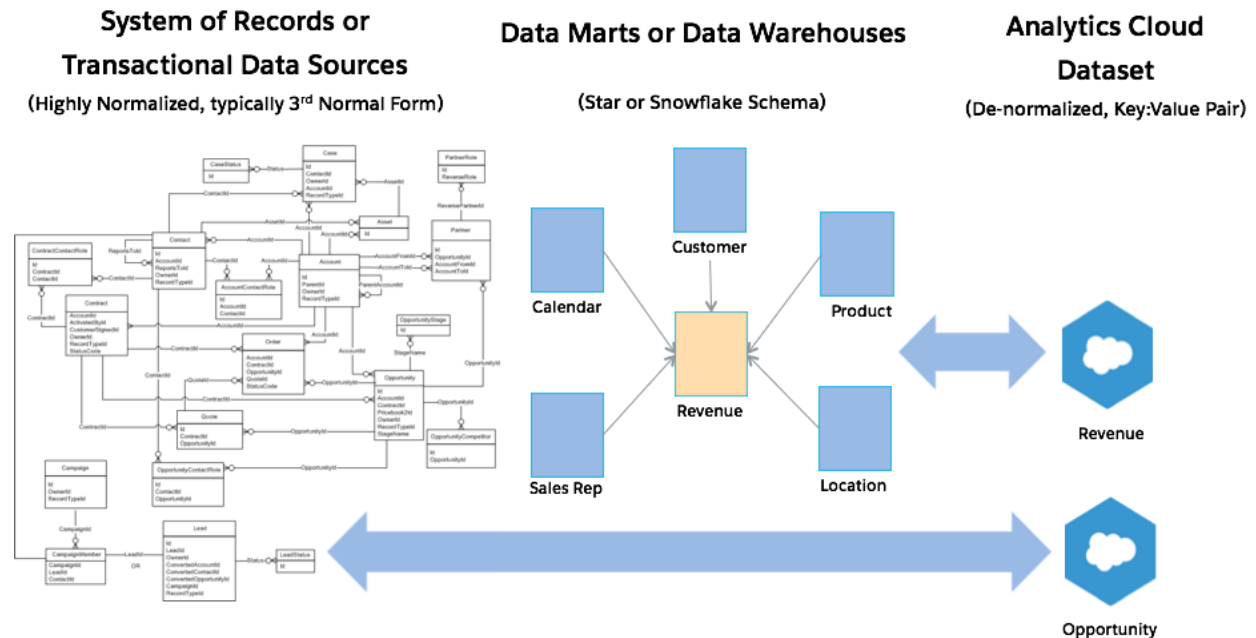


The basic unit of consumption of data in Wave Analytics Cloud is a Lens. A lens could be a single query step with its visualization widgets or a compilation of steps with corresponding widgets. The later is often referred to as a dashboard.

Before we dig deeper into SAQL, let's understand the different terminologies we use in Wave.

## Dataset

A dataset contains a set of source data, specially formatted and optimized for interactive exploration. Data is stored as denormalized records in key-value pairs.



Wave dataset has a schema-free design, enabling them to scale with new dimensions or measures without disrupting existing applications.

### RDBMS/ Structured Storage

ID	Name	SIC	Value
X0001	Company1	523	
X0002	Company2	399	Winter
X0003	Company3	425	
X0004	Company4		
X0005	Company5		
X0006	Company6	123	

### Key-Value Pair Storage

ID:X0001 Name:Company1 SIC:523
ID:X0002 Name:Company2 SIC:399 Value:Winter
ID:X0003 Name:Company3 SIC:425
ID:X0004 Name:Company4
ID:X0005 Name:Company5
ID:X0006 Name:Company6 SIC:123

Data is pushed into Wave through an ELT (Extract, Load, and Transform) process from Salesforce objects or external data loaded as CSV files or using a partner ETL tool.

A number of transformations can be applied to data during the ELT process. Details about all the available transform operations are available [here](https://help.salesforce.com/apex/HTViewHelpDoc?id=bi_integrate_tranformation_overview.htm&language=en_US)<sup>1</sup>.

<sup>1</sup>[https://help.salesforce.com/apex/HTViewHelpDoc?id=bi\\_integrate\\_tranformation\\_overview.htm&language=en\\_US](https://help.salesforce.com/apex/HTViewHelpDoc?id=bi_integrate_tranformation_overview.htm&language=en_US)

## Anatomy of a Wave Dashboard

Wave dashboards are created using the Builder<sup>2</sup> and stored as JSON documents in the Salesforce core system objects. Every dashboard contains a state node and other identifying metadata.

Dashboards are a simple collection of

### Metadata

Metadata contains identifying metadata like dashboard and application ID<sup>3</sup>, Owner information, Datasets used, etc.

### State

State node stores the dashboard logic. It includes the query definitions along with display configuration.

#### Steps

- Steps define the data elements
- Steps can take two forms
  - a compact form
  - SAQL (or PigQL) form

Compact Step	SAQL Step
<pre> "OpportunityName_2": {   "type": "aggregate",   "query": {     "measures": [       {         "sum",         "amount"       }     ],     "groups": [       "Opportunityname"     ]   },   "isGlobal": false,   "isFacet": true,   "useGlobal": true,   "selectMode": "single",   "start": null,   "datasets": [     {       "name": "Opportunity"     }   ],   "visualizationParameters": {     "visualizationType": "hbar"   } }</pre>	<pre> "OpportunityName_2": {   "selectMode": "single",   "query": {     "pigql": "q = load 'Opportunity'; q = group q by 'Opportunityname'; q = foreach q generate 'Opportunityname' as 'Opportunityname', sum('amount') as 'sum_amount'; q = limit q 2000;"   },   "measures": [     ["sum", "amount"]   ],   "groups": [ "Opportunityname" ] }, "isGlobal": false, "isFacet": true, "useGlobal": true, "selectMode": "single", "start": null, "datasets": [   {     "name": "Opportunity"   } ], "visualizationParameters": {   "visualizationType": "hbar" } }</pre>

<sup>2</sup> Builder refers to the Wave dashboard builder tool in Wave Analytics Cloud

<sup>3</sup> Every application, dashboard, lens and dataset gets a unique Salesforce identifier

- Both forms of steps get converted to a SAQL query by Wave runtime and submitted to the engine for execution. The response is passed to an associated widget that dictates the presentation of a chart or a data element.
- Supports dynamic mustache templates<sup>4</sup> for binding steps together

## Widgets

- Widgets define the display characteristics applied for steps
- Positioning in desktop runtime.
- e.g. widgets include a chart, text, number, compare table, etc.

**Widgets**

```

"chart_1": {
  "type": "chart",
  "position": {
    "h": 320,
    "w": 450,
    "x": 20,
    "y": 560,
    "zIndex": 90
  },
  "parameters": {
    "step": "step_report_AvgRunTime",
    "visualizationType": "time"
  }
},

```

## Layouts

- Layouts define the placement of Widgets on a dashboard.
- As of Winter'16 release<sup>5</sup>, only mobile app supports layouts.

**Layouts**

```

"layouts": [
  {
    "device": "iphone",
    "pages": [
      {
        "rows": [
          "box_25(colspan=1)",
          "Mobile_text_DashboardHeader(colspan=3) | row : (height=70)",
          "mobile_number_header_DBRT {colspan=2}",
          "mobile_number_1_DashboardFailures {colspan=2}",
          "mobile_text_header3 {colspan=2} |",
          "mobile_text_1_Dashboards {colspan=2}",
          "box_line4_Mobile {colspan=4 , hpad= -10,zIndex=-1} |",
          "row : (height=8)",
          "text_1_DashboardAvgRunTime(colspan=4, vpad=5)",
          "chart_1_AvgRunTime {colspan=4}",
          "box_line3_Mobile {colspan=4 , hpad= -10} | row :",
          "text_1_AvgDashboardComponentRunTime(colspan=4, vpad=5)",
          "chart_1_AvgRunTimeDashboardComponent {colspan=4}"
        ]
      }
    ],
    "version": 1
  }
],

```

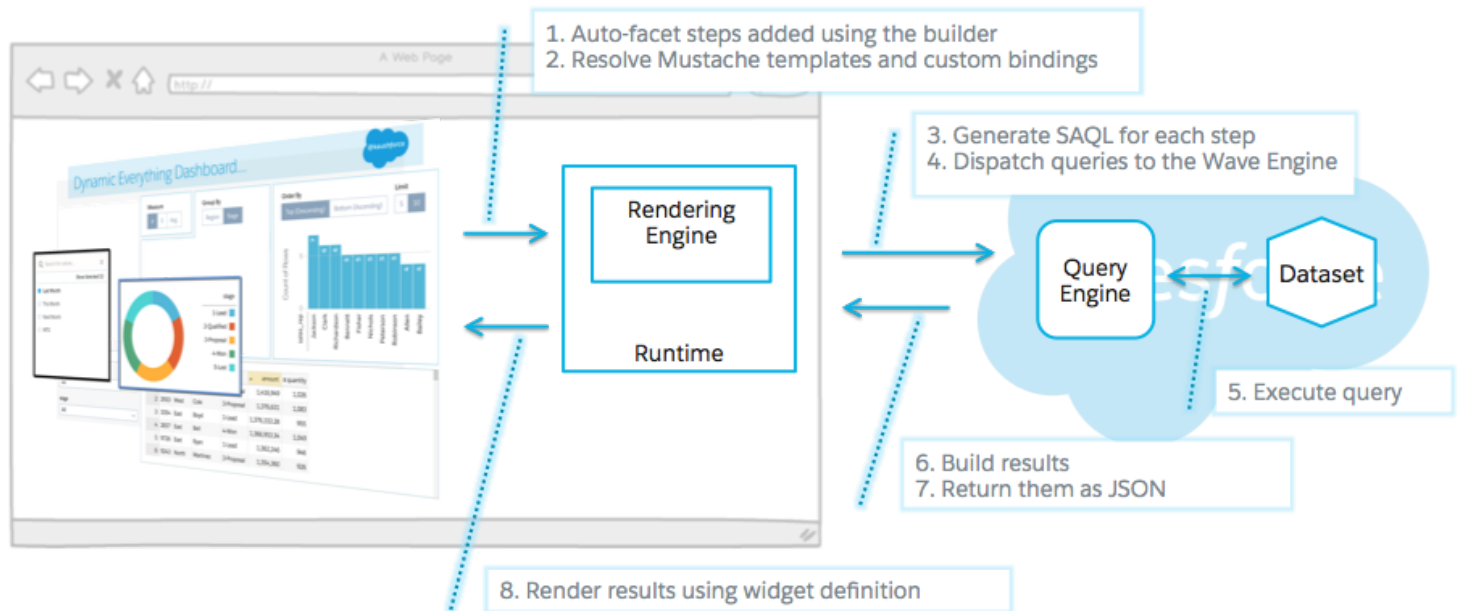
<sup>4</sup> Mustache templates allow output of one step to be injected into another step

<sup>5</sup> Salesforce Winter'16 release came out in October 2015



## Wave Runtime

Wave runtime (browser and mobile app) evaluates the steps, widgets, and layouts to render the final dashboard. It auto-facets the compact steps and resolves bindings and templates. Every step is converted to a SAQL query that gets to the engine for execution. The resulting data is passed to the charting library that renders it using corresponding widget definition.



More information about dashboard elements is available [here](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_json.meta/bi_dev_guide_json/)<sup>6</sup>

<sup>6</sup> [https://developer.salesforce.com/docs/atlas.en-us.bi\\_dev\\_guide\\_json.meta/bi\\_dev\\_guide\\_json/](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_json.meta/bi_dev_guide_json/)

# SAQL Basics

## Apache Pig-Latin

There is a growing need for ad-hoc analysis of extremely large data volumes as the collection and analysis of data has grown enormously. The sheer size of these data dictates that it be stored and processed on highly parallel systems. With this volume, the usual way of analyzing data writing imperative scripts is overly restrictive and time-consuming process that leads to more procedural programming.

A Pig Latin program is a sequence of steps, much like in a programming language, each of which carries out a single data transformation.

## Salesforce Analytics Query Language (SAQL)

**Salesforce Analytics Query Language (SAQL)** is the official query language of Wave Analytics Cloud. Almost every action you take in the Analytics Cloud results in one or more SAQL queries. Every lens, dashboard, and explorer action generates and executes a SAQL statement to build the data needed for the visualization. Official SAQL guide is available [here](#)<sup>7</sup>

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

Developers can write SAQL to access Analytics Cloud data via:

### Wave REST API

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

### Dashboard JSON

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

---

<sup>7</sup> [https://developer.salesforce.com/docs/atlas.en-us.bi\\_dev\\_guide\\_saql.meta/bi\\_dev\\_guide\\_saql/](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_saql.meta/bi_dev_guide_saql/)

## Elements of a SAQL Query

Elements of a SAQL query are defined in the official documentation [here](#)<sup>8</sup>.

A SAQL query loads an input dataset, operates on it, and outputs a results dataset. A query is made up of statements. Each SAQL statement has an input stream, an operation, and an output stream. A statement is made up of keywords (such as filter, group, and order), identifiers, literals, and special characters. Statements can span multiple lines and must end with a semicolon.

Each query line is assigned to a mandatory named stream. A named stream can be used as an 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.

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

## Windowing

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

Basic Windowing Syntax:

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

Example for this:

```
sum(sum(SalesAmount)) over([..] partition by all order by (Year asc, Month asc)) as
'Sum_SalesAmount'
```

The first part of the above aggregation, sum(SalesAmount), looks a lot like any other aggregation. Adding OVER designates it as a window function. You could read the above aggregation as “take the sum of SalesAmount over the entire result set, in order by Year ascending.

The order of execution of this query would be , It orders the result set by Year and Month ascending and does the aggregation sum(sum(SalesAmount)) with what is mentioned in Over (in this case it does the running sum) and reset the running sum by partition (In here, it will not reset, as the partition is by all).

<sup>8</sup> [https://developer.salesforce.com/docs/atlas.en-us.bi\\_dev\\_guide\\_saql.meta/bi\\_dev\\_guide\\_saql/](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_saql.meta/bi_dev_guide_saql/)

## Use Cases

In your dashboards, SAQL queries can be used for custom calculations. These calculations can involve measures from multiple streams from a single dataset or multiple datasets.

## Pros and Cons

SAQL is a useful tool to have when building dashboards, but should be used appropriately. When used inappropriately, it's use can prolong dashboard creation process and can complicate dashboard changes.

Pros	Cons
<b>Flexibility</b> Provides ability to derive new measures <b>Power</b> Ability to use engine functions that are not yet available in the UI <b>Control</b> Ability to co-group across datasets Ability to specify discrete filter conditions	<b>Maintenance</b> New syntax that requires programming skills <b>Limited exploration</b> Currently, SAQL based charts on a dashboard can not be opened in explorer for ad-hoc analysis

# SAQL Examples

## Derived Measures

A derived measure is a metric calculated at run-time, and its definition is contained in a single SAQL statement. For Example, if we have a 'close date' and a 'created date' for an Opportunity in a dataset, and we wanted to do "Time to Win", then "Time to Win" can be created as a Derived measure. An easy way to do the calculation would be to use the system generated day\_epoch measures.

```
q = load "Opp_for_SAQL";
q = foreach q generate 'CloseDate', 'CreatedDate', 'Name',
('CloseDate_day_epoch' - 'CreatedDate_day_epoch') as 'Time_to_Win';
q = group q by ('CloseDate', 'CreatedDate', 'Name');
q = foreach q generate 'CloseDate', 'CreatedDate', 'Name', avg ('Time_to_Win')
as 'Time_to_Win';
q = order q by 'CreatedDate';
```

	↕ CreatedDate	↕ CloseDate	↕ Time to Win
	2012-05-28	2013-02-02	250
ion	2012-06-03	2013-02-02	244
	2012-06-04	2013-02-02	243
..c.)	2012-06-04	2013-02-02	243
LLC	2012-06-07	2013-02-12	250
	2012-06-08	2013-02-12	249

## Derived Dimensions

These dimensions are projected in the SAQL query but not part of the dataset. Typically, used to bin data or group static text. For example, If we would like to concatenate two different dimensions to form a single dimension.

```
q = load "Stock";
q = group q by ('Name', 'Symbol');
q = foreach q generate 'Name' as 'Name', 'Symbol' as 'Symbol', 'Name' + " (" +
'Symbol' + ")" as 'Company and Symbol', count() as 'count';
q = limit q 2000;
```

Company and Symbol	Name	Symbol
SandRidge Mississippian Trust I (SDT )	SandRidge Mississippian Trust I	SDT
Six Flags Entertainment Corporation New (SIX )	Six Flags Entertainment Corporation New	SIX
Symantec Corporation (SYMC )	Symantec Corporation	SYMC
S aratoga Investment Corp (SAQ )	S aratoga Investment Corp	SAQ
S chweitzer-Mauduit International, Inc. (SWM )	S chweitzer-Mauduit International, Inc.	SWM

In this case, we are concatenating Stock Symbol with Company name to derive a dimension “Company and Symbol”. Binning will be covered later in this document.

## Bindings

When a user makes a selection on a widget in a dashboard, that selection value can be used to update other steps and widgets to make the dashboard interactive. This action is referred to as faceting. When enabled, all widgets and compact form steps on a dashboard that are linked by the same dataset are automatically bound to each other.

SAQL steps are not faceted by default and have to be explicitly bound. Also, if you have widgets on a dashboard from multiple datasets, they need to be bound to each other using Wave binding options.

On a high-level, there are two kinds of bindings:

- Results Binding - Pass the entire resultset from a step to another step
- Selection Binding - Pass the user selected values from a step to another step

## Binding Operations

Binding functions pass an array of values from the first step to the second step. For e.g. ["Central", "East", "West"] or [2015, 2016]. A few binding operations are provided to extract and manipulate values passed using selection or result binding.

Operation	Description
value( )	The value() function is used to get a single value from an array returned by a binding. By default, it returns the very first value from the array with quotes " "
single_quote()	<p>The single_quote( ) operation is typically used to format a "group" and "foreach generate" lines in the query.</p> <p>For example: "Owner.Name" converts to 'Owner.Name', and ["Owner.Name", "Owner.Region"] converts to ('Owner.Name', 'Owner.Region'). If this function is not used then the value is returned in double quotes.</p>

no_quote()	The no_quote() operation is typically used to correctly format the "order" line in a query. For example, ["desc"] converts to desc.
field()	The field() operation creates a field for each object in an array. Useful when a step is bound to another static step.

Let's review the binding types and these core bindings in context to the following example.

This example creates a dashboard with three widgets

1. A toggle selector to filter data by region using the opportunity dataset
2. A static toggle selector to choose the measure to select from opportunity product
3. A bar chart to represent product breakdown by revenue from opportunity product Dataset

### Dashboard Step for Defining the Query

The dashboard steps are defined below for each of these widgets

#### Source step

A pillbox/ toggle selector is used to choose regions to review. A user can pick one or more regions to see top accounts and products statistics.

```
"step_Region": {
  "selectMode": "multi",
  "query": {
    "measures": [["count", "*"]],
    "groups": ["Acct.Region"]
  },
  "isGlobal": false,
  "isFacet": true,
  "useGlobal": true,
  "selectMode": "single",
  "start": [
    "Central"
  ],
  "datasets": [
    {
      "name": "OpportunityDataset"
    }
  ],
  "visualizationParameters": {
    "visualizationType": "hbar"
  }
}
```

Here is what the above steps look like when assigned to a pillbox/toggle widget.

Region



### SQL query generated by the runtime to source data for the toggle widget

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

### Response data in JSON format from the Query Engine:

```
[{"Region": "Central", "count": 70},
{"Region": "East", "count": 106},
{"Region": "West", "count": 121}]
```

The binding functions extract data from this JSON response and pass it to other steps.

### Metric selector to display for Opportunity Product

To articulate binding operations, we'll also create a metric selector pillbox that allows a user to select the measure to use for Product i.e. List Price or Final Price

```
"step_measure": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "$ Total Price",
      "value": [{ "measure_value": "TotalPrice" }]
    },
    {
      "display": "$ List Price",
      "value": [{ "measure_value": "ListPrice" }]
    }
  ],
  "start": [
    [{ "measure_value": "TotalPrice" } ]
  ],
  "isGlobal": false,
  "useGlobal": false,
  "type": "static",
  "isFacet": false
}
```

Here is what the above steps look like when assigned to a pillbox/toggle widget.





## Destination Step from Opportunity Product connected with Opportunity using binding

```

"step_Product": {
  "selectMode": "single",
  "query": {
    "measures": [
      [
        "sum",
        "Price"
      ]
    ],
    "pigql": "q=load \"OpptyProduct\"; q = group q by 'Product.Name'; q =
filter q by 'OpptyAcct.Region' in {{selection(step_Region)}}; q = foreach q
generate 'Product.Name' as 'Name', sum('Price') as 'sum_Price'; q = order q by
'sum_Price' desc; q = limit q 2000;",
    "groups": [
      "Product.Name"
    ],
    "order": [
      [
        -1,
        { "ascending": false }
      ]
    ]
  },
  "visualizationParameters": {
    "visualizationType": "hbar"
  },
  "datasets": [
    {
      "name": "OpptyProduct"
    }
  ],
  "isGlobal": false,
  "useGlobal": true,
  "type": "aggregate",
  "isFacet": true
}

```

## Resulting SAQL query

```

q = load "OpptyProduct";
q = group q by 'Product.Name';
q = filter q by 'OpptyAcct.Region' in ["Central"];
q = foreach q generate 'Product.Name' as 'Product.Name', sum('Price') as
'sum_Price';
q = order q by 'sum_Price' desc;
q = limit q 2000;

```

The above step had pre-defined measure 'Price'. What if there were two different measures for price i.e. List Price or Total Price.

An important note about selection bindings is that the binding will return `all` and not `null` if there is no selection.

### Destination Step from Opportunity Product with dynamic measure selector

```
"step_Product": {
  "selectMode": "single",
  "query": {
    "measures": [
      [
        "sum",
        "Price"
      ]
    ],
    "pigql": "q=load \"OpptyProduct\"; q = group q by 'Product.Name'; q
= filter q by 'OpptyAcct.Region' in {{selection(step_Region)}}; q =
foreach q generate 'Product.Name' as 'Name', sum( {{
single_quote(value(field(selection(step_measure), 'measure_value'))}} )
as 'sum_Price'; q = order q by 'sum_Price' desc; q = limit q 2000;",
    "groups": [
      "Product.Name"
    ]
  },
  "visualizationParameters": {
    "visualizationType": "hbar"
  },
  "datasets": [
    {
      "name": "OpptyProduct"
    }
  ],
  "isGlobal": false,
  "start": null,
  "useGlobal": true,
  "type": "aggregate",
  "isFacet": true
}
```

The above step will let users choose the measure based on toggle selector with default value as 'TotalPrice'.

### Resulting SAQL query

```
q = load "OpptyProduct";
```

```
q = group q by 'Product.Name';
q = filter q by 'OpptyAcct.Region' in ["Central"];
q = foreach q generate 'Product.Name' as 'Product.Name', sum('TotalPrice') as
'sum_Price';
q = order q by 'sum_Price' desc;
q = limit q 2000;
```

The use of various binding operations will extract the following from these raw results.

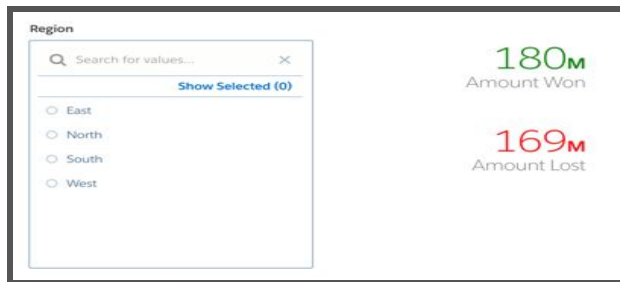
Operation	Description
q = filter q by 'OpptyAcct.Region' in {{ <b>selection</b> (step_Region) }}	<b>“Central”</b> for single selection; <b>[“Central”, “East”]</b> for multiple selections; <b>all</b> for no selection.
q = filter q by 'OpptyAcct.Region' in {{ <b>results</b> (step_Region) }}	<b>[“Central”, “East”, “West”]</b> . Result binding uses to full results from the source step and do not pass any selections made interactively on the region back to step_Product
q = filter q by 'OpptyAcct.Region' == {{ <b>value</b> ( <b>selection</b> (step_Region)) }}	<b>“Central”</b> <i>assuming central region is selected and</i> "selectMode": "singlerequired" in step_Region
q = filter q by 'OpptyAcct.Region' == \"{{ <b>no_quote</b> ( <b>value</b> ( <b>selection</b> (step_Region) )) }}\";	<b>Central</b> <i>assuming central region is selected. The example is for illustration of functionality and will only work properly with "selectMode": "singlerequired". Good examples for using this to control ascending/ descending sort is provided later in the document. You can also use no_quotes to pass numeric values into a query from a dimension field.</i>
{{ <b>single_quote</b> (value(field(selection(step_ measure), 'measure_value')) ) }}	<b>!ListPrice!</b> Assumes "selectMode": "singlerequired" in step_measure
{{value(field(selection(step_measure), 'mea sure_value')) }}	<b>!ListPrice!</b> Assumes "selectMode": "singlerequired" in step_measure. Notice that by default a selection returns value in quotes.

More information about binding operations with examples can be found [here](#)<sup>9</sup>

## Results Binding

Results binding is used to filter a step by using the **values** that result from another step. This type of binding is typically used across multiple datasets and is useful because the underlying steps that are being bound will automatically be faceted so you will not have to worry about manually applying filters. For example, consider the dashboard below.

We are displaying “Amount Won” and “Amount Lost” based on a region. If we want to calculate Win/Loss %, then we would have to take the result of “Amount Won” and “Amount Lost” and derive the Win/Loss %. Results binding typically involves multiple round trips to the server and, therefore, can have performance implications.



Our pigql step with results binding will look like,

### SAQL query with results binding

```
q = load \"Oppty\";
q = group q by all;
q = foreach q generate ({{ value(results(step_win)) }} /
  ({{ value(results(step_lost)) }} + {{ value(results(step_win)) }})) * 100
  as 'win_percent';
```

### Binded steps

```
"step_win": {
  "type": "aggregate",
  "query": {
    "measures": [
      "sum",
      "Amount"
    ],
    "filters": [
      "IsClosed",
```

<sup>9</sup>[https://developer.salesforce.com/docs/atlas.en-us.bi\\_dev\\_guide\\_json/meta/bi\\_dev\\_guide\\_json/bi\\_dbjson\\_binding\\_operations.htm](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_json/meta/bi_dev_guide_json/bi_dbjson_binding_operations.htm)

```

        ["true"],
        "in"],
    ["IsWon",
    ["true"],
    "in"]
    ]
},
"isGlobal": false,
"isFacet": true,
"useGlobal": true,
"selectMode": "single",
"start": null,
"datasets": [{"name": "Oppty"}],
"visualizationParameters": {"visualizationType": "hbar"}
},
"step_lost": {
    "type": "aggregate",
    "query": {
        "measures": [
            ["sum",
            "Amount"]
        ],
        "filters": [
            ["IsClosed",
            ["true"],
            "in"]
        ]
    },
    "isGlobal": false,
    "isFacet": true,
    "useGlobal": true,
    "selectMode": "single",
    "start": null,
    "datasets": [{"name": "Oppty"}],
    "visualizationParameters": {"visualizationType": "hbar"}
}

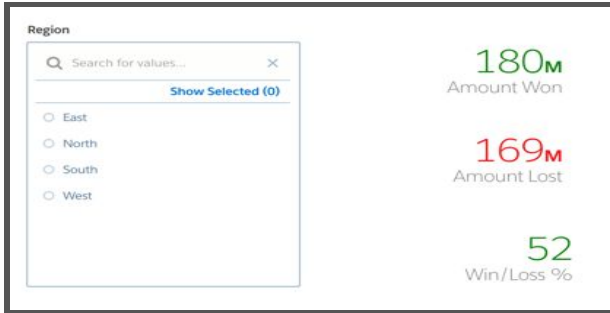
```

### Resulting SAQL query

```

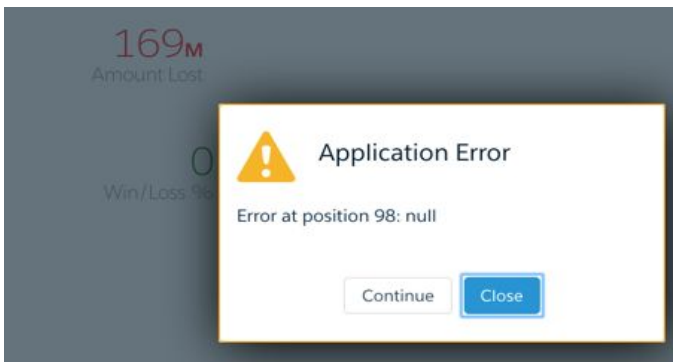
q = load "Oppty";
q = group q by all;
q = foreach q generate (180331550.95/(168515445.74 + 180331550.95)) * 100 as
'win_percent';

```



## Null handling in Results Binding

In the above dashboard, If any of the filters we selected returns 0 records for the Amount Won or Amount Lost steps then the result binding in the above query will error out:



### Resulting SAQL query

```
q = load "Oppty";
q = group q by all;
q = foreach q generate (null / (168515445.74 + null)) * 100 as
'win_percent';
```

The Coalesce() function can be used to handle nulls generated during the evaluation of a query by the engine on the server. However, since results binding can generate nulls on the client side, one can use the following approach to avoid errors.

### SAQL query with results binding

```
q = load "Oppty";
q = group q by all;
q = foreach q generate coalesce({{value(results(step_win))}} / ({
value(results(step_lost))}} + {{value(results(step_win))}}),0) * 100 as
'win_percent';
```

Most of the heavy lifting for this needs to be done in the step\_win and step\_lost steps. We must make sure that these steps will never return no value. An easy way to do this is to turn

these steps into SAQL with autoFilter turned on and add a value of 0. The SAQL of step\_win (and step\_lost) should look something like the following.

```
q = load "Oppty";
-- autoFilter filters are injected here
-- Example autoFilter filter that would cause the dashboard to fail.
q = filter q by 'StageName' == "Closed Lost";
-- Will return no records because these filters are contradictory
q = filter q by 'IsWon' == "true";
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';

r = load "Oppty"; -- Must load the dataset again. Cannot use the q stream
because it has already been filtered with the autoFilter and there will be no
records in it. autoFilter filters are loaded after the first load statement so
they will not be applied to this load statement.
r = group r by all; -- Notice that this group is going off of the unfiltered q
stream
r = foreach r generate 0 as 'sum_Amount'; -- Ensuring that 'sum_Amount' will
have at least one value. Think of this as a pseudo coalesce to 0

u = union q, r;
u = group u by all;
u = foreach u generate sum('sum_Amount') as 'sum_Amount';
```

Remember to turn on [autoFilter](#)<sup>10</sup> in the “query” parameter.

### Resulting SAQL query

```
q = load "Oppty";
q = group q by all;
q = foreach q generate coalesce(0 / (168515445.74 + 0), 0) * 100 as
'win_percent';
```

### **Null Handling Alternative without Bindings**

There are certain instances when bindings are unnecessary and thus we can handle nulls directly with coalesce. For example, we can rewrite the same “win\_percent” query without the use of bindings and still have the step faceted to the rest of the dashboard.

### SAQL query

```
q = load "Oppty";
q = filter q by 'IsClosed' == "true";
```

---

<sup>10</sup>

[https://developer.salesforce.com/docs/atlas.en-us.bi\\_dev\\_guide\\_json.meta/bi\\_dev\\_guide\\_json/bi\\_dbjson\\_query.htm](https://developer.salesforce.com/docs/atlas.en-us.bi_dev_guide_json.meta/bi_dev_guide_json/bi_dbjson_query.htm)

```
w = filter q by 'IsWon' == "true";
u = group w by all, q by all;
u = foreach u generate coalesce(sum(w['Amount'])/sum(q['Amount']), 0) as
'win_percent';
```

We would not be able to take advantage of auto faceting if our query uses two separate datasets. This is because auto facet filters are inserted immediately after first load statement and are only associated with the dataset specified in the step's (not the query's) dataset parameter. An example of this is if we are trying to compute the percent of our quota that's been attained.

Assuming:

```
"datasets": [{"name": "Oppty"}]
```

### SAQL query

```
o = load "Oppty";
-- autoFilter filters for Oppty are placed here
o = filter o by 'IsWon' == "true";
q = load "Quota";
-- autoFilter filters are not placed here or anywhere else in the query
u = group w by all, q by all;
u = foreach u generate coalesce(sum(w['Amount'])/sum(q['Amount']), 0) as
'win_percent';
```

In the example, auto facet filters for the Oppty dataset stream “o” will be applied correctly because it is the first dataset loaded; however, filters the Quota dataset stream “q” will not be auto faceted because it is not available specified in the dataset parameter. (If Quota was the first dataset loaded, Oppty auto facet filters would be applied to the Quota dataset, which is most likely not what you want.)

For this case, use bindings and null handling as shown in the previous section.

### Using Intermediate steps in Results Binding

Intermediate steps are mostly used in results binding. These steps are not associated with a widget but used to filter another step that is associated with a widget. For example, let's say we want to display opportunities for top 5 products, where Product and Opportunities are two different datasets.

In this case, we will have an intermediate step “Top5Products” from the product dataset and use that results in the opportunity dataset to filter out the opportunities.

### Source step

```
"Top5Products": {
```



```

    "selectMode": "single",
    "query": {
      "measures": [
        [
          "sum",
          "Amount"
        ]
      ],
      "limit": 5,
      "groups": [
        "ProductName"
      ],
      "order": [
        [
          -1,
          {
            "ascending": false
          }
        ]
      ]
    },
    "datasets": [
      {
        "name": "ProductDataset"
      }
    ],
    ...
  },

```

We then use the results from the above query to get total opportunity amount.

### SAQL query with selection binding

```

q = load \"OpportunityDataset\";
q = filter q by 'Product' in {{results(Top5Products)}};
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';"

```

### Resulting SAQL query

```

q = load \"OpportunityDataset\";
q = filter q by 'Product' in in ["Product A1", "Product B2", "Product C3",
"Product D4", "Product E5"];
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';"

```

## Selection Binding

When a user makes a selection from a widget on a dashboard, that selected value can be used to update other steps and widgets to make the dashboard interactive. Almost all parts of a pigql can include a selection binding to the results of a prior query. In an aggregate query, the fields that can include a selection binding are:

- Group
- Measure
- Filters
- Sort
- Limit

We will be using a “Static Step” to explain selection binding. Static steps populate a list of static values, instead of results from a query. For example, If we want to give the users flexibility to change the sort order or to limit the number of rows returned, we could use static step to provide those choices. Ensure that every static step on your dashboard returns a default value. This default value loads at dashboard runtime and avoids any step failure.

## Dynamic Sort Order

Usage: To let dashboard users pick a sort order.

We are going to have a static step that defines a sort order. On selection of that Sort order, we are going to dynamically change the sort order of an another step.

### Source step

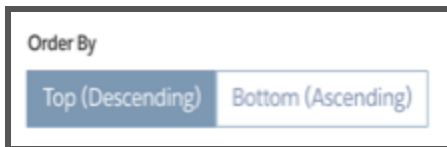
```
"step_sort_direction": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "Top (Descending)",
      "value": [
        {
          "sortorder": "desc"
        }
      ]
    },
    {
      "display": "Bottom (Ascending)",
      "value": [
        {
          "sortorder": "asc"
        }
      ]
    }
  ]
}
```

```

],
  "dimensions": []
  "start": [
    [
      {
        "sortorder": "desc"
      }
    ]
  ],
  "isGlobal": false,
  "useGlobal": false,
  "type": "static",
  "isFacet": false
},

```

This is what the above steps looks like when it is assigned to a pillbox/toggle widget.

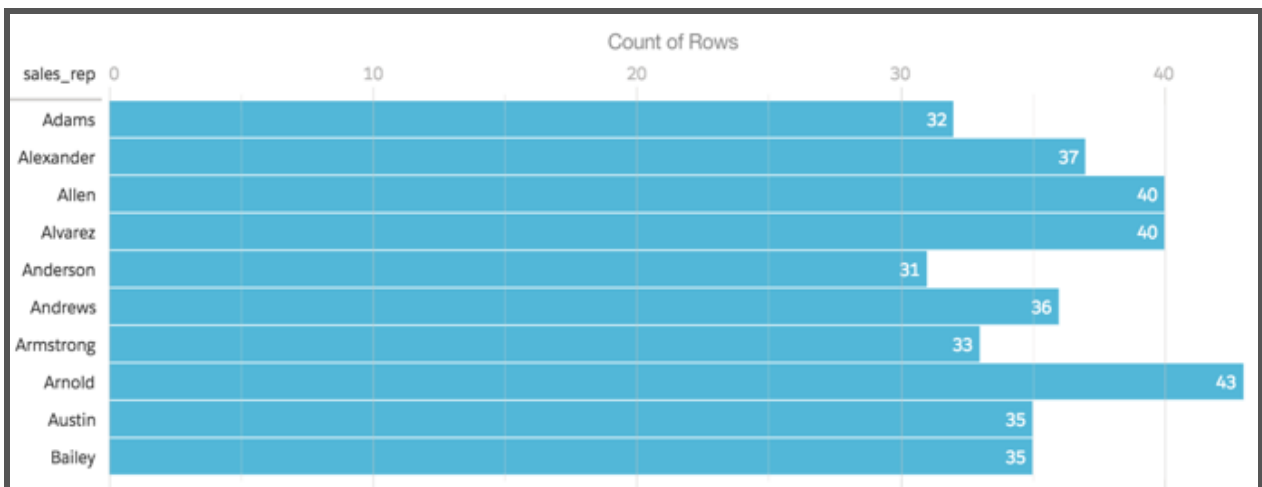


### SAQL query before selection binding

```

q = load "Dataset";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';

```



### SAQL query with selection binding

```

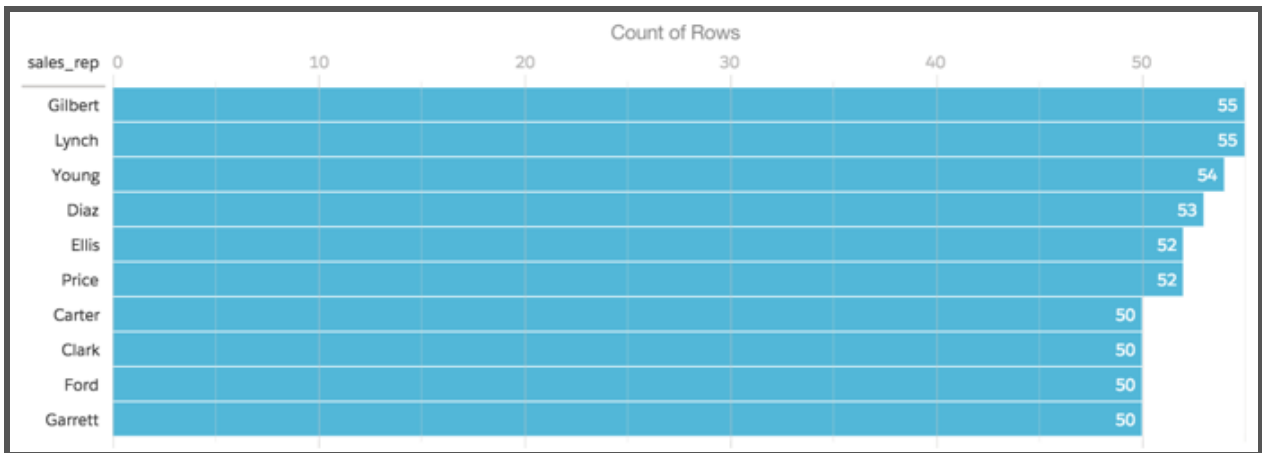
q = load "Dataset";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';

```

```
q = order q by 'count' {{
no_quote(value(field(selection(step_sort_direction),'sortorder')))) }};
```

### Resulting SAQL query

```
q = load "Dataset";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';
q = order q by 'count' desc;
```



### Dynamic Row Limits

Usage: To let dashboard users decide how many rows are returned.

We are going to have a static step that defines row limit and on the selection of that row limit we are going to dynamically limit the number of rows displayed. All static steps must provide a default start value to avoid a query error when dashboard loads for the first time.

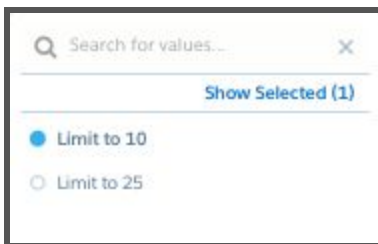
### Source Step

```
"step_limit": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "Limit to 10",
      "value": [
        {
          "limit": 10
        }
      ]
    },
    {
      "display": "Limit to 25",
```

```

        "value": [
            {
                "limit": 10
            }
        ]
    },
    "start": [
        [
            {
                "limit": 25
            }
        ]
    ],
    "dimensions": [],
    "isGlobal": false,
    "useGlobal": false,
    "type": "static",
    "isFacet": false
},

```



### SAQL query before selection binding

```

q = load "datest";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';
q = order q by 'count' desc;

```



### SAQL query with selection binding

```
q = load "datest";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';
q = order q by 'count' desc;
q = limit q by {{ no_quote(value(field(selection(step_limit),limit))) }};
```

### Resulting SAQL query

```
q = load "datest";
q = group q by 'sales_rep';
q = foreach q generate 'sales_rep' as 'sales_rep', count() as 'count';
q = order q by 'count' desc;
q = limit q by 10;
```



### **Dynamic Filter (String Dimension)**

Usage: To let dashboard users filter a step based on selections from another step that uses the same dataset, a different dataset or even a static step.

Lets have a look at some simple filtering using a static step. We are going to use a static step that emits an opportunity stage upon selection; this selection will then be used to dynamically filter opportunities returned by another step.

### Source Step

```
"step_filter": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "Lost",
      "value": "5-Lost"
    },
    {
      "display": "Won",
      "value": "4-Won"
    },
    {
      "display": "Proposal",
      "value": "3-Proposal"
    },
    {
      "display": "Qualified",
      "value": "2-Qualified"
    },
    {
      "display": "Lead",
      "value": "1-Lead"
    }
  ],
  "start": [
    "1-Lead"
  ],
  "dimensions": []
  "isGlobal": false,
  "useGlobal": false,
  "type": "static",
  "isFacet": false
}
```

Search for values...

Show Selected (1)

☐ Lost  
☐ Won  
☐ Proposal  
☐ Qualified  
☒ Lead

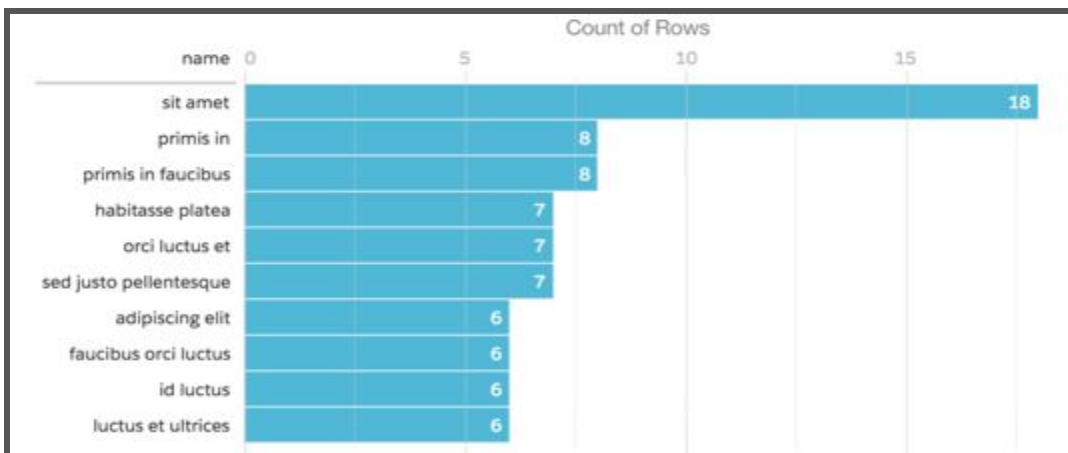
### SAQL query with selection binding

```
q = load \"Dataset\";
q = filter q by 'stage' in {{selection(step_filter)}};
q = group q by 'name'; q = foreach q generate 'name' as 'name', count() as 'count';
q = order q by 'count' desc;
q = limit q 50;
```

Since the static step has a default value of “1-Lead”, Only the records with Stage = Lead will be returned.

### Resulting SAQL query

```
q = load \"Dataset\";
q = filter q by 'stage' in [\"1-Lead\"];
q = group q by 'name'; q = foreach q generate 'name' as 'name', count() as 'count';
q = order q by 'count' desc;
q = limit q 50;
```





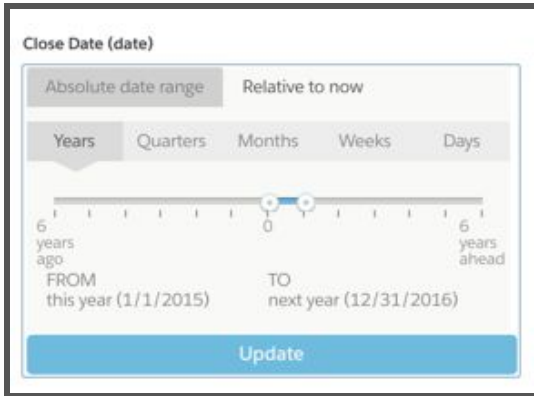
## Dynamic Filter (Full Date)

Here is a way to use a date widget grouped by a Year, Month and Day to dynamically filter another step that returns opportunities from a dataset.

### Source Step

```
"CloseDate_Year_CloseDate_Month_CloseDate_Day_1": {
  "selectMode": "single",
  "query": {
    "measures": [
      [
        "count",
        "*"
      ]
    ],
    "groups": [
      [
        "CloseDate_Year",
        "CloseDate_Month",
        "CloseDate_Day"
      ]
    ]
  },
  "datasets": [
    {
      "name": "Opp_Fiscal"
    }
  ],
  "visualizationParameters": {
    "visualizationType": "hbar"
  },
  "isGlobal": false,
  "start": [
    [
      [
        "year",
        0
      ],
      [
        "year",
        1
      ]
    ]
  ],
}
```

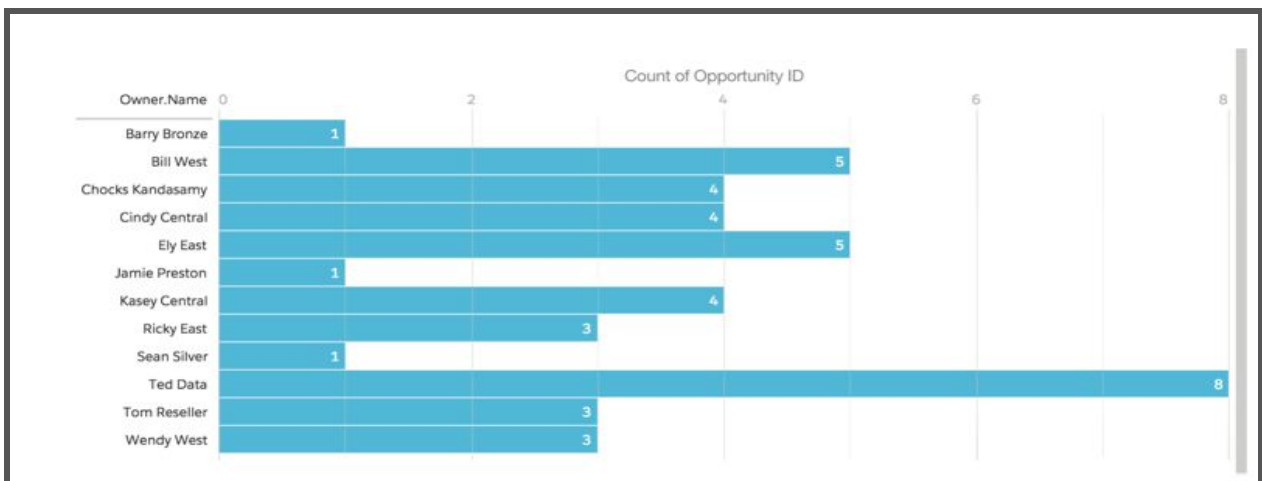
```
"useGlobal": true,
"type": "aggregate",
"isFacet": true
}
```



The image shows a Salesforce date range picker interface. It has two tabs: 'Absolute date range' (selected) and 'Relative to now'. Below the tabs are five buttons: 'Years', 'Quarters', 'Months', 'Weeks', and 'Days'. A horizontal timeline shows a range from '6 years ago' to '6 years ahead', with a central '0' marker. Below the timeline, it specifies 'FROM this year (1/1/2015)' and 'TO next year (12/31/2016)'. An 'Update' button is at the bottom.

### SAQL query before selection binding

```
q = load \"Opp_Fiscal\";
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('Id') as
'unique_Id';
```



### SAQL query with selection binding

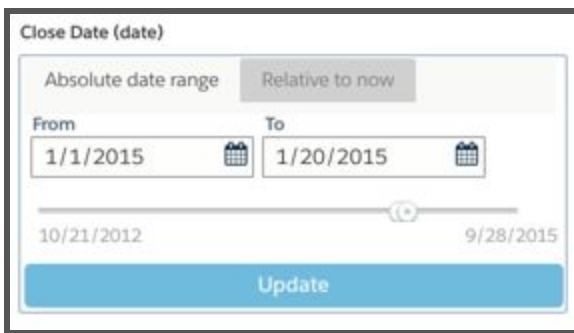
```
q = load \"Opp_Fiscal\";
q = filter q by date('CloseDate_Year','CloseDate_Month','CloseDate_Day') in
{{selection(CloseDate_Year_CloseDate_Month_CloseDate_Day_1)}};
q = group q by 'Owner.Name';
```

```
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('unique_Id') as
'unique_Id';
```

### Resulting SAQL query (if current year is selected)

```
q = load \"Opp_Fiscal\";
q = filter q by date('CloseDate_Year','CloseDate_Month','CloseDate_Day') in
[\"current year\"..\"current year\"];
q = group q by 'Owner.Name' ;
q = foreach q generate 'Owner.Name' as 'Owner.Name',unique('Id') as
'unique_Id';
```

And if we selected “Absolute Date”( Let’s say Jan 01 to Jan 20), the resulting query will be



### Resulting SAQL query (if Jan 1, 2015 to Jan 20, 2015 is selected)

```
q = load \"Opp_Fiscal\";
q = filter q by date('CloseDate_Year','CloseDate_Month','CloseDate_Day') in
[dateRange([2015,1,1], [2015,1,20])];
q = group q by 'Owner.Name' ;
q = foreach q generate 'Owner.Name' as 'Owner.Name',unique('Id') as
'unique_Id';
```

## Dynamic Filter (Partial Date)

Here is a way to use a step grouped by a partial date to dynamically filter another step that returns opportunities from a different dataset. To use the emitted partial date in the target step for filtering, we have to project these date fields.

### Source Step

```
"CloseDate_Year_Fiscal_CloseDate_Quarter_Fiscal_1": {
  "selectMode": "single",
  "query": {
    "measures": [
      [
        "count",
        "*"
      ]
    ]
  }
}
```

```

    ]
  ],
  "groups": [
    [
      "CloseDate_Year_Fiscal",
      "CloseDate_Quarter_Fiscal"
    ]
  ]
},
"datasets":
[
  {
    "name": "Opp_Fiscal"
  }
],
"visualizationParameters":
{
  "visualizationType": "hbar"
},
"isGlobal": false,
"start": null,
"useGlobal": true,
"type": "aggregate",
"isFacet": true
},

```

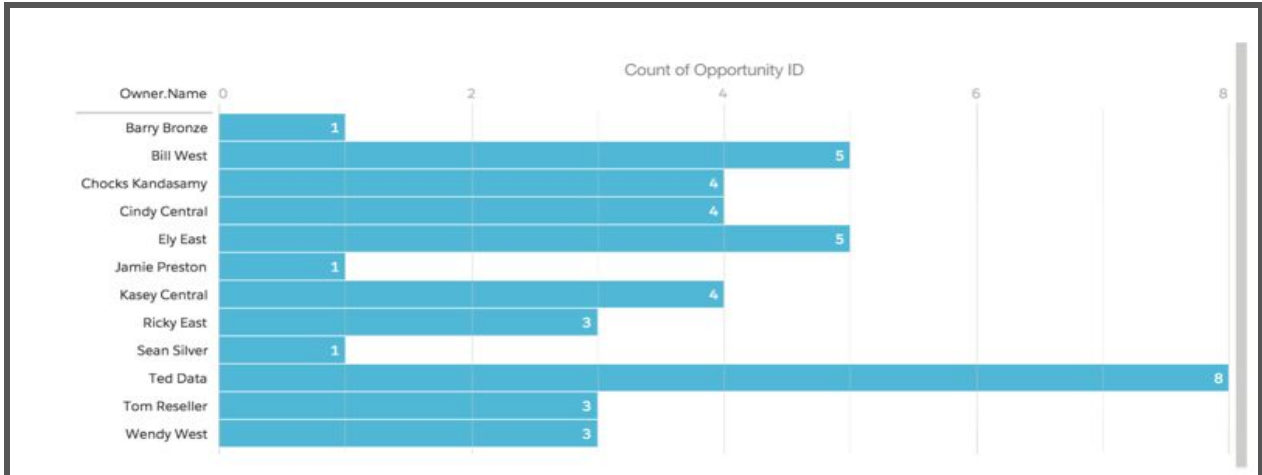


### SAQL query before selection binding

```

q = load \"Opp_Fiscal\";
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('Id') as
'unique_Id';

```



### SAQL query with selection binding

We have to project the date dimension before they can be used in a filter. The above query will look like this after selection:

```
q = load \"Opp_Fiscal\";
q = group q by ('CloseDate_Year_Fiscal',
'CloseDate_Quarter_Fiscal','Owner.Name');
q = foreach q generate 'CloseDate_Year_Fiscal' + \"~~~\" +
'CloseDate_Quarter_Fiscal' as
'CloseDate_Year_Fiscal~~~CloseDate_Quarter_Fiscal','Owner.Name' as
'Owner.Name', unique('Id') as 'unique_Id';
q = filter q by 'CloseDate_Year_Fiscal~~~CloseDate_Quarter_Fiscal' in
{{selection(CloseDate_Year_Fiscal_CloseDate_Quarter_Fiscal_1)}};
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('unique_Id') as
'unique_Id';
```

### Resulting SAQL query (if 2013 - 1 is selected)

```
q = load \"Opp_Fiscal\";
q = group q by ('CloseDate_Year_Fiscal',
'CloseDate_Quarter_Fiscal','Owner.Name');
q = foreach q generate 'CloseDate_Year_Fiscal' + \"~~~\" +
'CloseDate_Quarter_Fiscal' as
'CloseDate_Year_Fiscal~~~CloseDate_Quarter_Fiscal','Owner.Name' as
'Owner.Name', unique('Id') as 'unique_Id';
q = filter q by 'CloseDate_Year_Fiscal~~~CloseDate_Quarter_Fiscal' in
[\"2013~~~1\"];
q = group q by 'Owner.Name';
```

```
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('unique_Id') as
'unique_Id';
```

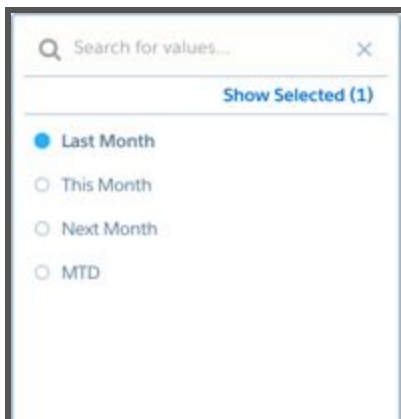
## Dynamic Filter (Relative Date)

Let's add a static step that emits a relative period that then gets used to dynamically filter another step that returns opportunities.

### Source Step

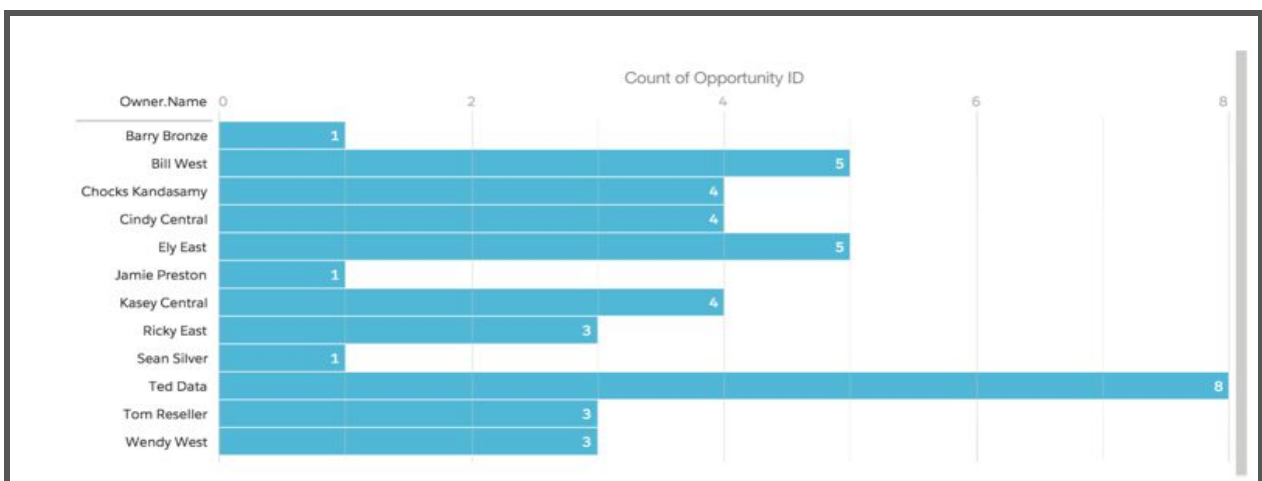
```
"step_relative_date": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "Last Month",
      "value": [
        {
          "step_value": "[\"1 month ago\"..\"1 month ago\"]"
        }
      ]
    },
    {
      "display": "This Month",
      "value": [
        {
          "step_value": "[\"current month\"..\"current month\"]"
        }
      ]
    },
    {
      "display": "Next Month",
      "value": [
        {
          "step_value": "[\"1 month ahead\"..\"1 month ahead\"]"
        }
      ]
    },
    {
      "display": "Month To Date",
      "value": [
        {
          "step_value": "[\"current month\"..\"current day\"]"
        }
      ]
    }
  ],
  "start": [
```

```
[
  {
    "step_value": "[\"1 month ago\"..\\"1 month ago\"]"
  }
],
"isGlobal": false,
"useGlobal": false,
"type": "static",
"isFacet": false
}
```



### SAQL query before selection binding

```
q = load \"Opp_Fiscal\";
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('Id') as
'unique_Id';
```



## SAQL query with selection binding

```
q = load \"Oppty\";
q = filter q by date('created_Year', 'created_Month', 'created_Day') in {{
no_quote(value(field(selection(step_relative_date), 'step_value'))) }};
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('Id') as
'unique_Id';
```

## Resulting SAQL query

```
q = load \"Oppty\";
q = filter q by date('created_Year', 'created_Month', 'created_Day') in [\"1
month ago\"..\"1 month ago\"];
q = group q by 'Owner.Name';
q = foreach q generate 'Owner.Name' as 'Owner.Name', unique('Id') as
'unique_Id';
```

## Dynamic Grouping

Usage: To dynamically change the grouping of a step results. For Example, Use this approach to display opportunities grouped by “Region” or by Opportunity “Stage”. Dashboard user decides how the results should pivot.

Let’s use a static step that emits “Stage” or “Region” (string) based on user’s selection. we will use the emitted string to dynamically change the group on another step that returns opportunities.

### Source Step

```
"step_group": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "Region",
      "value": [
        {
          "group": "region"
        }
      ]
    },
    {
      "display": "Stage",
      "value": [
        {
          "group": "stage"
        }
      ]
    }
  ]
}
```



```

    ]
  }
],
"start": [
  [
    {
      "group": "stage"
    }
  ]
],
"isGlobal": false,
"useGlobal": false,
"type": "static",
"isFacet": false
},

```



### SAQL query with selection binding

```

q = load \"Oppty\";
q = group q by {{ single_quote(value(field(selection(step_group), 'group')) ) }};
q = foreach q generate {{
single_quote(value(field(selection(step_group), 'group')) ) }}, count() as
'count';
q = limit q 2000;

```

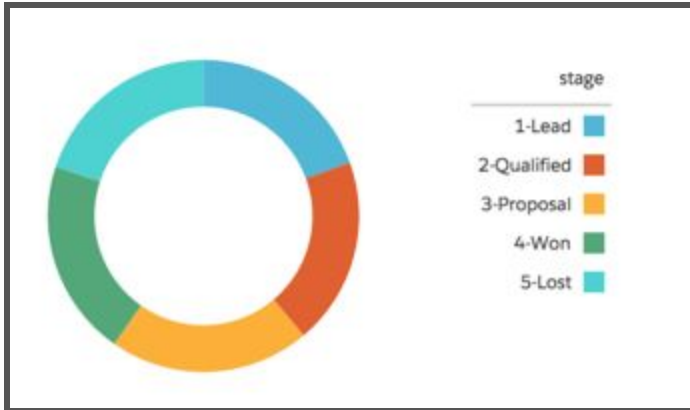
For a chart to plot a grouped resultset, the query must project (generate) the group value. We therefore add the same binding to the `group by` as well as `generate` statements.

### Resulting SAQL query

```

q = load \"Oppty\";
q = group q by 'stage';
q = foreach q generate 'stage', count() as 'count';
q = limit q 2000;

```



## Dynamic Measure

Usage: To dynamically change the measure of a widget. For Example, use this approach to display opportunity count, opportunity amount or by average amount and let dashboard user pick one.

Let's use a static step that emits a SAQL fragment on selection, we will dynamically add this fragment to another step that returns opportunities with the selected measure.

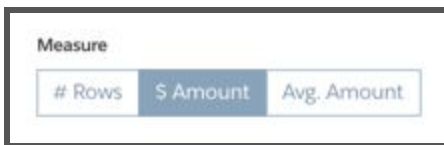
### Source Step

```
"step_measure": {
  "selectMode": "singlerequired",
  "values": [
    {
      "display": "# Rows",
      "value": [
        {
          "saql_step_value": "count() as 'Final_Result'"
        }
      ]
    },
    {
      "display": "$ Amount",
      "value": [
        {
          "saql_step_value": "sum(amount) as 'Final_Result'"
        }
      ]
    },
    {
      "display": "Avg. Amount",
      "value": [
        {
          "saql_step_value": "avg(amount) as 'Final_Result'"
        }
      ]
    }
  ]
}
```

```

    }
  ]
}
],
"start": [
  [
    {
      "saql_step_value": "count() as 'Final_Result'"
    }
  ]
],
"isGlobal": false,
"useGlobal": false,
"type": "static",
"isFacet": false
}

```



### SAQL query with selection binding

```

q = load \"Oppty\";
q = group q by 'stage';
q = foreach q generate 'stage',
{{no_quote(value(field(selection(step_measure), 'saql_step_value')))}};

```

### Resulting SAQL query

```

q = load \"Oppty\";
q = group q by 'stage';
q = foreach q generate 'stage', sum(amount) as 'sum_amount';

```



## Binding Summary

Any section of a query can be dynamically generated using selection binding. Here is a consolidated example.

### SAQL query with selection binding

```
q = load \"Oppty\";
q = filter q by date('created_Year', 'created_Month', 'created_Day') in
{{no_quote(value(field(selection(step_relative_date), 'step_value')))}};
q = group q by {{single_quote(value(field(selection(step_group), 'group')))}};
q = foreach q generate
{{single_quote(value(field(selection(step_group), 'group')))}},
{{no_quote(value(field(selection(step_measure), 'saql_step_value')))}};
q = order q by 'count'
{{no_quote(value(field(selection(step_sort_direction), 'sortorder')))}};
q = limit q by
{{no_quote(value(field(selection(step_sort_direction), limit))}};
```

### Target step

```
"step_name":
  "query": {
    "pigql": "q = load \"Oppty\";
q = filter q by date('created_Year', 'created_Month', 'created_Day') in
{{no_quote(value(field(selection(step_relative_date), 'step_value')))}};
q = group q by {{single_quote(value(field(selection(step_group), 'group')))}};
q = foreach q generate
{{single_quote(value(field(selection(step_group), 'group')))}},
{{no_quote(value(field(selection(step_measure), 'saql_step_value')))}};
q = order q by 'count'
{{no_quote(value(field(selection(step_sort_direction), 'sortorder')))}};
```

```
q = limit q by
{{no_quote(value(field(selection(step_sort_direction),limit))}}};",
    "measures": [["Final","Result"]],
    "groups": [
        "{{value(field(selection(step_group),'group'))}}"
    ]
},
"selectMode": "single",
"datasets":
[
    {
        "name": "Oppty"
    }
],
"visualizationParameters":
{
    "visualizationType": "hbar"
},
"start": null,
"isGlobal": false,
"useGlobal": true,
"type": "aggregate",
"isFacet": true
}
```

## Field-to-Field Filters

There are situations where you may want to filter your data based on how values from different fields compare. For example, we may only want to see

- opportunities where the account owner and opportunity owner are the same person;
- opportunities that were opened and closed in the same month;
- opportunities whose 'Adjusted\_Amount' is smaller than its original 'Amount'

If you want to filter using a filter statement (as opposed to using a cogroup or case statement), the filter statement must be in a specific location for these types of filters. The data stream must be projected before you can use this type of filter 'Account\_Owner' == 'Opportunity\_Owner'. This is because pre-projection filters must have a constant value on the right hand side.

Here are a few examples:

### Dimensions

Say we want to find the number of opportunities that an account owner has where they are also the opportunity owner.

### Does not work:

```
q = load "Opportunity";
q = filter q by 'Account_Owner' == 'Opportunity_Owner';
q = group q by ('Account_Owner', 'Opportunity_Owner');
q = foreach q generate 'Account_Owner', 'Opportunity_Owner', count() as
'count';
```

This query does not work because the filter statement occurs before the data stream is ever projected.

### Works:

```
q = load "Opportunity";
q = group q by ('Account_Owner', 'Opportunity_Owner');
q = foreach q generate 'Account_Owner', 'Opportunity_Owner', count() as
'count';
q = filter q by 'Account_Owner' == 'Opportunity_Owner';
```

While the filter statement is exactly the same as the previous query, this query works because the filter statement is now written after a projection.

### Works, but not optimal:

```
q = load "Opportunity";
q = foreach q generate 'Account_Owner', 'Opportunity_Owner';
q = filter q by 'Account_Owner' == 'Opportunity_Owner';
q = group q by ('Account_Owner', 'Opportunity_Owner');
q = foreach q generate 'Account_Owner', 'Opportunity_Owner', count() as
'count';
```

This query works because the first projection occurs before the filter. It will return the same output as the previous query, but it is not optimal because there are two projections and the first projection does not affect the data (it is only there so that we can use the filter).

A slightly different use case...we want to find the total number of opportunities that have the same account owner as opportunity owner.

```
q = load "Opportunity";
q = foreach q generate 'Account_Owner', 'Opportunity_Owner';
q = filter q by 'Account_Owner' == 'Opportunity_Owner';
q = group q by all;
q = foreach q generate count() as 'count';
```

For this use case, it is not possible to avoid the extra projection because, unlike the previous use case, the fields used in the filter are not being used in the group statement. We have to filter before the aggregation.

## Dates

A common use case around comparing different dates is identifying opportunities that were opened and closed in the same month. Field-to-field filters makes it relatively easy to do this.

```
q = load "Opportunity";
q = foreach q generate 'Created_Date_Year', 'Created_Date_Month',
'Closed_Date_Year', 'Closed_Date_Month';
q = filter q by 'Created_Date_Year' == 'Closed_Date_Year';
q = filter q by 'Created_Date_Month' == 'Closed_Date_Month';
q = group q by all;
q = foreach q generate count() as 'count';
```

Again, you'll need to be mindful of where you place your field-to-field filters (i.e., after the first projection).

## Measures

Another use case for field-to-field filters is when comparing measures such as when we want to get opportunities that have an adjusted amount that is less than the original amount (assuming there is no field in the object that records that difference). This use case has several workarounds, some which actually do not require field-to-field filters.

First here are two queries that will not work:

### Does not work:

```
q = load "Opportunity";
q = filter q by 'Amount' > 'Adjusted_Amount';
q = group q by 'Opportunity_Id';
q = foreach q generate 'Opportunity_Id', sum('Amount') as 'sum_Amount',
sum('Adjusted_Amount') as 'sum_Adjusted_Amount';
```

This query does not work because the filter statement occurs before the data stream is ever projected.

### Does not work:

```
q = load "Opportunity";
q = filter q by ('Amount' - 'Adjusted_Amount') > 0;
q = group q by 'Opportunity_Id';
q = foreach q generate 'Opportunity_Id', sum('Amount') as 'sum_Amount',
sum('Adjusted_Amount') as 'sum_Adjusted_Amount';
```

Even though the filter in this query does have a constant on the right side, it is still not valid because there are two fields on the left side. `( 'Amount' - 'Adjusted_Amount' ) > 0` is equivalent to `'Amount' > 'Adjusted_Amount'`.

The following three queries are valid:

#### Works:

```
q = load "Opportunity";
q = group q by 'Opportunity_Id';
q = foreach q generate 'Opportunity_Id', sum('Amount') as 'sum_Amount',
sum('Adjusted_Amount') as 'sum_Adjusted_Amount';
q = filter q by 'sum_Amount' > 'sum_Adjusted_Amount';
```

While the filter statement is exactly the same as query above, this query works because the filter statement is now written after a projection.

#### Works:

```
q = load "Opportunity";
q = group q by 'Opportunity_Id';
q = foreach q generate 'Opportunity_Id', sum('Amount') as 'sum_Amount',
sum('Adjusted_Amount') as 'sum_Adjusted_Amount';
q = filter q by ('sum_Amount' - 'sum_Adjusted_Amount') > 0;
```

We are also able to do field calculations in the filter statement if it occurs after a projection.

#### Works:

```
q = load "Opportunity";
q = group q by 'Opportunity_Id';
q = foreach q generate 'Opportunity_Id', sum('Amount') as 'sum_Amount',
sum('Adjusted_Amount') as 'sum_Adjusted_Amount', sum('Amount') -
sum('Adjusted_Amount') as 'sum_Adjustment';
q = filter q by 'sum_Adjustment' > 0;
```

In this query, we are able to avoid field-to-field filters altogether by simply calculating the difference in the prior statement and using the new field to filter.

## Calculations across multiple data streams (datasets)



There are some scenarios where you may want to perform a calculation using values from multiple data streams. These streams can be from the same or different datasets. There are a few ways to do this in SAQL.

Additional documentation on [co-group](#) and [union](#).

## Joins vs. Co-group

Unlike relational database joins, where the base records from multiple streams/objects come together; Wave co-groups result in aggregate joins. First, all the streams (datasets) are aggregated (or rolled-up) on one or more dimensions; and then these aggregated rows get joined with each other. Co-groups can provide significant performance advantages over traditional joins.

## Sample Data

We will use the Opportunity and Quota datasets in the following sections.

### Opportunity

Opportunity			
Owner	Amount	IsClosed	IsWon
Phil	\$ 10,000	Yes	Yes
Phil	\$ 12,000	Yes	Yes
Phil	\$ 20,000	Yes	No
Phil	\$ 20,000	No	No
Kaush	\$ 13,000	Yes	Yes
Kaush	\$ 10,000	Yes	Yes
Chocks	\$ 10,000	Yes	Yes
Chocks	\$ 9,000	Yes	Yes
Justin	\$ 10,000	Yes	No

### Quota

Quota	
Name	Amount
Phil	\$ 50,000
Kaush	\$ 25,000
Justin	\$ 10,000

## Inner Co-Group

An inner co-group means that two input streams, called left and right are grouped independently and arranged side by side. Only data that exists in both groups appears in the results

Example: You want to calculate the percent quota attainment for all your sales reps off of your opportunity and quota datasets.

```
opt = load "Opportunity";
quo = load "Quota";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
att = group opt by 'Owner', quo by 'Name';
att = foreach att generate opt['Owner'] as 'Owner',
sum(opt['Amount'])/sum(quo['Amount']) as 'sum_attainment_perc';
```

Output:

Inner Co-Group	
Owner	sum_attainment_perc
Phil	0.44
Kaush	0.92

You'll notice that neither Chocks nor Justin appears in the output of the inner co-group because they did not appear in both the Opportunity and Quota datasets. Hence, it does not matter which name field we use (opt['Owner'] or quo['Name']) since only names that appear in both data streams are in the output. However, this will not be the case for left, right, and full co-groups

## Left Outer Co-Group with Coalesce

A left outer co-group combines the right data stream to the left data stream. If a record on the left side does not have a match to the right, the missing right value will come through as null, and the unmatched left record will not be dropped.

Coalesce replaces a null value with a specified value when the query is evaluated on the Wave server.

Example: Same as the inner co-group example except if a sales rep does not have any quota data, you want to set the default quota value to 10,000.

```
opt = load "Opportunity";
quo = load "Quota";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
att = group opt by 'Owner' left, quo by 'Name';
att = foreach att generate opt['Owner'] as 'Owner',
sum(opt['Amount'])/sum(coalesce(quo['Amount'], 10000)) as
'sum_attainment_perc';
```

Output:

Left Outer Co-Group	
Owner	sum_attainment_perc
Phil	0.44
Kaush	0.92
Chocks	1.9

We actively chose to use `opt['Owner']` rather than `quo['Name']` in the `generate` statement. If we had used `quo['Name']`, Chocks' closed won amount would have come through but his name would not since it does not exist in `quo['Name']`. Additionally, the `coalesce` function is needed in this example to prevent the query from erroring out on Chocks' attainment calculation. `10000/null` is not valid.

## Right Outer Co-Group with Coalesce

A right outer co-group combines the left data stream to the right data stream. If a record on the right side does not have a match to the left, the missing left value will come through as null, and the unmatched right record will not be dropped.

`Coalesce` replaces a null value with a specified value.

Example: Same as the left outer co-group example except if a sales rep does not have any closed won opportunity data, you want to set the default quota value to 0.

```
opt = load "Opportunity";
quo = load "Quota";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
att = group opt by 'Owner' right, quo by 'Name';
att = foreach att generate quo['Name'] as 'Owner', sum(coalesce(opt['Amount'],
0))/sum(quo['Amount']) as 'sum_attainment_perc';
```

Output:

Right Outer Co-Group	
Owner	sum_attainment_perc
Phil	0.44
Kaush	0.92
Justin	0.00

We actively chose to use `quo['Name']` rather than `opt['Owner']` in the `generate` statement. If we had used `opt['Owner']`, Justin's closed won amount would have come through, but his name would not since it does not exist in `opt['Owner']`. Additionally, the `coalesce` function is needed in this example to prevent the query from erroring out on Justin's attainment calculation. `null/10000` is not valid.

## Full Outer Co-Group with Coalesce

A full outer co-group combines the left and right data streams. If any records on either side do not match to the other side, the record is not dropped, and the missing value will come through as null.

Coalesce replaces a null value with a specified value.

Example: A combination of the left outer and right outer co-group examples.

```
opt = load "Opportunity";
quo = load "Quota";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
att = group opt by 'Owner' full, quo by 'Name';
att = foreach att generate opt['Owner'] as 'opt_Owner', quo['Name'] as
'quo_Name', sum(coalesce(opt['Amount'], 0))/sum(coalesce(quo['Amount'], 10000))
as 'sum_attainment_perc';
```

Output:

Full Outer Co-Group		
Owner	Name	sum_attainment_perc
Phil		0.44
Kaush		0.92
Chocks		1.90
	Justin	0.00

If you do not use a `coalesce` in this example, your query will try to perform a calculation with null values (null/10000 and 10000/null) and it error out.

## Co-Group as Filter

Co-groups can also function as filters. Suppose I want to find the accounts that have more than 100 critical or high priority cases, that have at least 1 opportunity, and the total amount of each of those accounts' opportunities. We need to use two separate datasets (Case and Opportunity) to get our answer. We would not be able to achieve this through a dataflow augment of the Case and Opportunity object because it would require a many-to-many join.

```
-- Get a list of accounts from the Case dataset that match the criteria we're
-- looking for. Filtering on priority and on number of cases for an account.
cas = load "Case";
cas = filter cas by 'Priority' in ["Critical", "High"];
cas = group cas by 'AccountId';
```

```
cas = foreach cas generate 'AccountId', count() as 'count';
cas = filter cas by 'count' > 100;
opp = load "Opportunity";
q = group opp by 'AccountId', cas by 'AccountId'; -- This inner cogroup
functions as a filter on 'AccountId'. Only accounts that exist in both the cas
and opp streams will be retained.
q = foreach q generate cas['AccountId'] as 'AccountId', sum('Amount') as
'sum_Amount';
```

## Union

Union operation allows us to combine 2 or more separate result streams into a single resultset. It is mandatory for the structure of both resultsets to match.

Example 1: In the full outer co-group example, we ended up with two name fields, and neither field captured every rep name in both the Opportunity and Quota datasets. The `union` operator allows us to create one name field that captures every name.

```
opt = load "Opportunity";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
opt = group opt by 'Owner';
opt = foreach opt generate 'Owner' as 'Owner', sum('Amount') as
'sum_Attainment';
quo = load "Quota";
quo = group quo by 'Name';
quo = foreach quo generate 'Name' as 'Owner', sum('Amount') as 'sum_Quota';
att = union opt, quo;
att = group att by 'Owner';
att = foreach att generate 'Owner' as 'Owner', sum(coalesce('sum_Amount',
0))/sum(coalesce('sum_Quota', 10000)) as 'sum_attainment_perc';
```

Output:

Union	
Owner	sum_attainment_perc
Phil	0.44
Kaush	0.92
Chocks	1.90
Justin	0.00

Example 2: You want to calculate the average opportunity amount for each sales rep as well as the average opportunity amount across the org.

```
opt = load "Opportunity";
opt = filter opt by 'IsClosed' == "Yes";
opt = filter opt by 'IsWon' == "Yes";
```

```
rep = group opt by 'Owner';
rep = foreach rep generate 'Owner' as 'Owner', sum('Amount') as 'sum_Amount';
all = group opt by all;
all = foreach rep generate "Total" as 'Owner', sum('Amount') as 'sum_Amount';
q = union rep, all;
```

Output:

Union	
Owner	sum_Amount
Phil	\$ 22,000
Kaush	\$ 23,000
Chocks	\$ 19,000
Total	\$ 64,000

Notice that in the `all` stream, we created a field called `'Owner'` (the same field name as the `'Owner'` field in the `rep` stream) and set the value of that field to be the string `"Total"`. This will create a totals row in the final results.

## Case Statement

### Binning

Binning is a technique used to group values (either dimensions or measures) into a smaller number of “bins”. In the following example, we bucket all records that have Type equals to “Services” or has word “New” as “New”. All the records that have word “Business” in the Type as “Old” and all other types are bucketed as “Other”. When this query executes, we get a new dimension called `Type_Bucket` that does not exist in the source dataset. Also, note that the order of the conditions is important. For example, the Type value “New Business” will be bucketed as “New Business” and not “Old Business” because the condition `'Type' matches "New " occurs before 'Type' matches "Business"`.

Opportunity:

Type	Amount
New Business	\$5,000
New Business	\$3,000
New Account	\$5,000
Services	\$4,000
New Business	\$7,000
Renewal	\$9,000
Add-On Business	\$6,000
Renewal	\$3,000
Renewal	\$9,000
Services	\$2,000
Renewal	\$2,000

```
q = load "Opportunity";
```

```
q = group q by ('Type', 'Type_Bucket');
q = foreach q generate (case
    when 'Type' matches "New " or 'Type' == "Services" then "New"
    when 'Type' matches "Business" then "Old Business"
    else "Other"
end) as 'Type_Bucket', 'Type', sum('Amount') as 'sum_Amount';
```

Before case statements were available, we used union queries to implement binning. Case statement significantly shortens and simplifies binning queries.

Output:

Type	Type_Bucket	Amount
New Business	New Business	\$15,000
New Account	New Business	\$5,000
Services	New Business	\$6,000
Add-On Business	Old Business	\$6,000
Renewal	Other	\$23,000

## Threshold

In this example, a visual indication is given when a threshold value is reached using the traffic light metaphor. The SAQL shown backs a donut chart with a grouped by dimension 'RAG' with various values Red, Amber, Green, which indicates the threshold segmentation reached. An XMD entry is used to assign the color to the dimension value. The key performance indicators are Total Actual Revenue (step shown below), Total Target and Total Percentage Achieved of revenue against the target; the SAQL makes use of those KPI steps when calculating the thresholds.

### Source step

```
"TotalActual__c_7": {
    "selectMode": "single",
    "query": {
        "measures": [
            [
                "sum",
                "TotalActual__c"
            ]
        ]
    },
    "datasets":
    [
        {
            "name": "Sales_Target"
        }
    ],
    "visualizationParameters":
```

```
{
  "visualizationType": "hbar"
},
{
  "isGlobal": false,
  "start": null,
  "useGlobal": true,
  "type": "aggregate",
  "isFacet": true
},

```

### SAQL query

```
q = load \"Sales_Target\";
q = group q by all;
q = foreach q generate
{{value(results(TotalActual__c_7))}}/{{value(results(TotalTarget__c_8))}}*100
as 'sum_Percent_Total_Acheived';
q =foreach q generate 'sum_Percent_Total_Acheived',
  (case when 'sum_Percent_Total_Acheived' >=100 then \"Green\"
    when 'sum_Percent_Total_Acheived' <100 &&
      'sum_Percent_Total_Acheived' > 60 then \"Amber\"
    else \"Red\"
  end ) as 'RAG' ;
```

The following needs to be added to the XMD file to set threshold colors created in the above query.

```
"colors": {
  "RAG": {
    "Red": "#FF3030",
    "Green": "#008B00",
    "Amber": "#FFD700"
  }
}
```

### Resulting SAQL query

```
q = load "Sales_Target";
q =group q by all;
q = foreach q generate (10075085.54/23417788.49)*100 as
'sum_Percent_Total_Achieved';
q =foreach q generate 'sum_Percent_Total_Achieved', case when
'sum_Percent_Total_Achieved' >=100 then "Green" when
'sum_Percent_Total_Achieved' <100 && 'sum_Percent_Total_Achieved' > 60 then
"Amber" else "Red" end as 'RAG';
```





## Time-based analysis

Time-based analysis relies heavily on date functions and ranges, which are documented [here](#). Some common examples are detailed below.

### MTD (Month To Date)

Calculate the total value of the opportunities expected to close from the beginning of the month through today.

```
q = load "Opportunity";
q = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day') in
["current month".."current day"];
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';
```

The relative date range filters for the beginning of the month through today. As the month or day changes, these keywords will refer to the current month and day.

### YTD (Year To Date)

Calculate the total value of the opportunities expected to close from the beginning of the year through today.

```
q = load "Opportunity";
q = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day') in
["current year".."current day"];
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';
```

### QTD (Quarter To Date)

Calculate the total value of the opportunities expected to close from the beginning of the quarter through today.

```
q = load "Opportunity";
q = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day') in
["current quarter".."current day"];
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';
```

### QoQ (Quarter over Quarter)

Calculate the percent change in total value of the opportunities expected to close for the current quarter compared to last quarter.

```
q = load "Opportunity";
curr = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["current quarter".."current quarter"];
prev = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["1 quarter ago".."1 quarter ago"];
QoQ = group curr by all, prev by all;
QoQ = foreach QoQ generate (sum(curr['Amount'])/sum(prev['Amount'])) - 1 as
'sum_QoQ';
```

Query Explanation: After loading in the opportunity dataset, we split it into two streams. One stream, `curr`, gets data for the current quarter using the date range `["current quarter".."current quarter"]` which filters for data ranging from the start of the current quarter to the end of the current quarter. The other stream, `prev`, gets data for the previous quarter. We then bring the two streams back together using a co-group. Since we are trying to calculate the overall QoQ number, we can group by `all` and then perform the QoQ calculation.

## Windowing Alternative

Let us see how we can do the same QoQ using Windowing.

```
q = load "Opportunity";
q = group q by ('CloseDate_Year', 'CloseDate_Quarter');
q = foreach q generate 'CloseDate_Year' + "~~~" + 'CloseDate_Quarter' as
'CloseDate_Year~~~CloseDate_Quarter', ((sum('Amount') / sum(sum('Amount'))
over([-1 .. -1] partition by all order by ('CloseDate_Year',
'CloseDate_Quarter')))) - 1) as 'sum_QoQ';
```

After loading the opportunity dataset, We are projecting Year and Quarter as we wanted to see the result in a Time Series and the formula we are using here is (Current Quarter/Previous Quarter) - 1. So if we have a close look at this particular line

```
((sum('Amount') / sum(sum('Amount')) over([-1 .. -1] partition by all order by
('CloseDate_Year', 'CloseDate_Quarter')))) - 1)
```

The order of execution will be to order the dataset by Year and Quarter ascending and the `sum('Amount')` which will give the Sum for the current quarter and `sum(sum('Amount'))` which will give us the sum for the previous quarter because `over(-1 .. -1)` indicates that we go one previous quarter and subtracting -1 at the end to get QoQ. Since the partitioning is by "all", windowing will happen across all Year-Quarter.

## QTD over QTD

Calculate the percent change in total value of the opportunities expected to close for the current quarter to date compared to last quarter to date.

```
q = load "Opportunity";
curr = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["current quarter".."current day"];
prev = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["1 quarter ago".."current day - 1 quarter"];
QoQ = group curr by all, prev by all;
QoQ = foreach QoQ generate (sum(curr['Amount'])/sum(prev['Amount'])) - 1 as
'sum_QoQ';
```

Query Explanation: This query utilizes techniques from the QTD and QoQ calculations. Similar to the QoQ query, we split the dataset into curr and prev streams; however, rather than filtering for the entire quarter, we filter for the quarter to date as we did in the QTD query. The way to get the same number of days from the previous quarter as the number of days elapsed in the current quarter is by using date math. "current day - 1 quarter" returns the current day relative to the previous quarter.

## Current QTD over last year same QTD

Calculate the percent change in total value of the opportunities expected to close for the current quarter to date compared to the same period last year.

```
q = load "Opportunity";
curr = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["current quarter".."current day"];
prev = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day')
in ["current quarter - 1 year".."current day - 1 year"];
QoQ = group curr by all, prev by all;
QoQ = foreach QoQ generate (sum(curr['Amount'])/sum(prev['Amount'])) - 1 as
'sum_QoQ';
```

Query Explanation: This query is similar to the QTD over QTD calculation. The only difference is that we refer to the previous year using relative date math ["current quarter - 1 year".."current day - 1 year"].

## Most recent available month

There are various reasons that your data could be out of date. For example, your bookings from an external data source may only get updated once a month on the 5th. On March 1-4, you still want to report off of January data. In these cases, standard relative dates may not be appropriate. Filtering for "previous month" on February 28th will give us our previous month's bookings data; however, on March 1st, the "previous month" filter will look for data for

February's bookings data, which does not get entered into the system until four more days. The work around for this is to use the following steps.

Opportunity:

Current Date = March 1, 2015		
CloseDate	Amount	CloseDate_day_epoch
01-23-2015	\$ 10,000	1421971200
01-31-2015	\$ 10,000	1422662400
02-07-2015	\$ 5,000	1423267200
02-08-2015	\$ 10,000	1423353600
02-21-2015	\$ 1,000	1424476800

```
q = load "Opportunity";
q = group by ('CloseDate_Year', 'CloseDate_Month');
q = foreach q generate 'CloseDate_Year', 'CloseDate_Month', sum('Amount') as
'sum_Amount';
q = order q by ('CloseDate_Year' desc, 'CloseDate_Month' desc);
q = limit q 1;
```

Output:

```
q = load "Opportunity";
q = group by ('CloseDate_Year', 'CloseDate_Month');
q = foreach q generate 'CloseDate_Year', 'CloseDate_Month', sum('Amount')
as 'sum_Amount';
```

CloseDate_year	CloseDate_month	Amount
2015	01	\$ 20,000
2015	02	\$ 16,000

```
q = order q by ('CloseDate_Year' desc, 'CloseDate_Month' desc);
```

CloseDate_year	CloseDate_month	Amount
2015	02	\$ 16,000
2015	01	\$ 20,000

```
q = limit q 1;
```

CloseDate_year	CloseDate_month	Amount
2015	02	\$ 16,000

Query Explanation: The objective of this query is to order the results by date in descending order and then select the first record to get the most recent date. This is achieved by ordering by the year and then month, both in descending order. The sequence in which you order is important here; for example, ordering by month and then year would give you different results.

Since we are trying to find the most recent month available, we apply a group at the year-month level. If we wanted the most recent available day, the grouping would be

( 'CloseDate\_year', 'CloseDate\_month', 'CloseDate\_day' ). For most recent available quarter, ( 'CloseDate\_year', 'CloseDate\_quarter' ).

If we were to try using standard relative dates like in the query below, the result would be \$0 because there is no data for the current month.

```
q = load "Opportunity";
q = filter q by date('CloseDate_year', 'CloseDate_month', 'CloseDate_day') in
["current quarter".."current quarter"];
q = group q by all;
q = foreach q generate sum('Amount') as 'sum_Amount';
```

Please note that \$0 is not the incorrect result for the query. Rather, the query itself does not address the question that is proposed.

## MoM on most recent available month

Calculate the percent change in total value of the opportunities expected to close for the most recent available month compared to the month before it.

```
q = load "Opportunity";
q = group q by ('CloseDate_Year', 'CloseDate_Month');
q = foreach q generate 'CloseDate_Year', 'CloseDate_Month', sum('Amount') as
'sum_Amount';
q = order q by ('CloseDate_Year' desc, 'CloseDate_Month desc');
curr = limit q 1;
prev = offset q 1;
prev = limit prev 1;
MoM = group curr by all, prev by all;
MoM = foreach MoM generate (sum(curr['sum_Amount'])/sum(prev['sum_Amount']))
- 1 as 'sum_MoM';
```

Query Explanation: This query builds upon the query to get the most recent available month. After ordering by year and month, we create two separate streams. `curr` represents the most recent available month and `prev` represents the preceding month. `curr` is the first record in the projection which is why we apply the limit of 1. We can get `prev`, which is the second record in the projection, by offsetting the projection by 1 and then getting the first record of this projection.

## Windowing Alternative

Let us see how we can do the same MoM on recent available month using Windowing.

```
q = load "Opportunity";
q = group q by ('CloseDate_Year', 'CloseDate_Month');
```

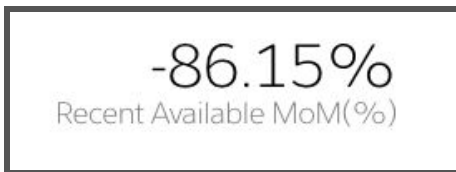
```
q = foreach q generate 'CloseDate_Year', 'CloseDate_Month', ((sum('Amount') /
sum(sum('Amount')) over([1 .. 1] partition by all order by ('CloseDate_Year'
desc, 'CloseDate_Month' desc))) - 1) * 100 as 'sum_MostRecentAvailableMoM';
q = order q by ('CloseDate_Year', 'CloseDate_Month');
q = limit q 1;
```

### Query Explanation:

```
((sum('Amount') / sum(sum('Amount')) over([1 .. 1] partition by all order by
('CloseDate_Year' desc, 'CloseDate_Month' desc))) - 1)
```

The order of execution will be to order the dataset by Year and Month ascending and the `sum('Amount')` which will give the Sum for the current month and `sum(sum('Amount'))` which will give us the sum for the previous month because `over([1 .. 1])` indicates that we go one month previous and subtracting -1 at the end to get MoM. Since the partitioning is by “all”, windowing will happen across all Year-Month. At the end since we are limiting the result set by 1, we will only get the Most recent available Month in the dataset.

Note that even though the windowing function contains the keywords “order by”, the resulting stream is not guaranteed to be ordered (the “order by” here is for the calculation). This is why there is a separate “order by” statement following the windowing function.



## Timeline Comparisons

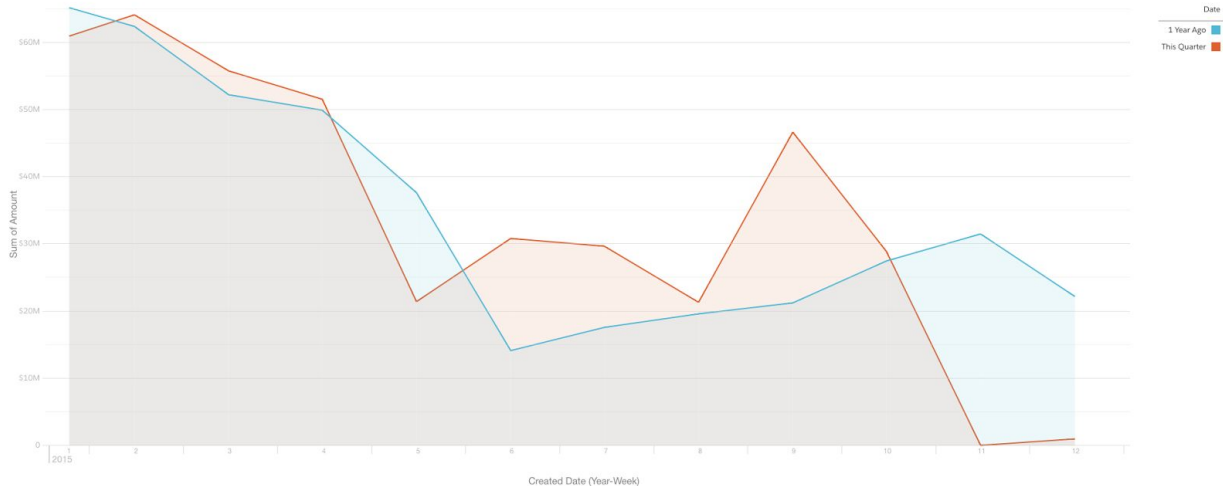
Another type of time-based analysis is comparing how we are trending in the current quarter versus the same time last year. This type of analysis is useful for seeing if we are on-track, lagging, or ahead by using a previous time period as a benchmark.

```
q = load "Opportunity";
ThisQ = filter q by date('CreatedDate_Year', 'CreatedDate_Month',
'CreatedDate_Day') in ["current quarter".."current quarter"];
LastQ = filter q by date('CreatedDate_Year', 'CreatedDate_Month',
'CreatedDate_Day') in ["current quarter - 1 year".."current quarter - 1 year"];
LastQGrp = cogroup ThisQ by (CreatedDate_Week) left, LastQ by
(CreatedDate_Week);
LastQGrp = foreach LastQGrp generate first(ThisQ['CreatedDate_Year']) + "~~~" +
ThisQ['CreatedDate_Week'] as 'CreatedDate_Year~~~CreatedDate_Week', "1 Year
Ago" as 'Date', sum(LastQ['Amount']) as 'sum_Amount';
ThisQ = group ThisQ by (CreatedDate_Year, CreatedDate_Week);
```

```

ThisQ = foreach ThisQ generate 'CreatedDate_Year' + "~~~" + 'CreatedDate_Week'
as 'CreatedDate_Year~~~CreatedDate_Week', "This Quarter" as 'Date',
sum('Amount') as 'sum_Amount';
c = union ThisQ, LastQGrp;

```



In this example we have two SAQL streams. One for this quarter's data which should hopefully be straightforward and one for the same quarter last year. To explain what we did with the 1 Year Ago stream, it's important to note that in order to use the time chart the final SAQL stream must be grouped by a valid date dimension and include the year and if we need to include the year we need to make sure the two separate data streams have the same year.

To manipulate the year in last year data stream, we cogrouped this quarter with 1 year ago using the CreatedDate\_Week dimension. In the projection, we took the year (we needed to use the first() function to get this since it wasn't used in my grouping) and month from the "This Quarter" stream and the sum of Amount from the "1 Year Ago" stream. This projection gives the count of created cases by week for the quarter one year ago but instead of my date field containing 2014 it contains 2015 so the data will show up on the time chart.

Finally we union the two streams together which are both grouped by year and month.

## Advanced Math Operations

With Spring '16 release, there are now new math and windowing functions available for more advanced analytics. These new functions allow you to perform some more advanced math operations that can be a powerful tool in improving your analytics.

Documentation for [math](#) and [windowing](#) functions.

## Correlation Coefficient

The correlation coefficient is useful for quantifying linear relationships between two fields (measures).

"The correlation coefficient ranges from -1 to 1. A value of 1 implies that a linear equation describes the relationship between X and Y perfectly, with all data points lying on a line for which Y increases as X increases. A value of -1 implies that all data points lie on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables."

[https://en.wikipedia.org/wiki/Pearson\\_product-moment\\_correlation\\_coefficient](https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient)

The formula to get the correlation coefficient is the following:

$$\text{correlation coefficient} = \frac{\text{covariance}_{x,y}}{\text{standard deviation}_x \times \text{standard deviation}_y}$$

where:  $\text{covariance}_{x,y} = \sum_{i=1}^N \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}$

$$\text{standard deviation}_x = \sqrt{\sum_{i=1}^N \frac{(x_i - \bar{x})^2}{N}}$$

### Query with comments:

```
q = load "Opportunity";
q = group q by 'Id';
q = foreach q generate 'Id',
    sum('Amount') as 'Amount', -- sum('Amount') does not change the value of
    'Amount' at a record level because 'Id' is a unique field and there is only 1
    'Amount' for each 'Id'. We are doing this sum() so that we still retain the
    'Amount'.
    sum('OpenDuration') as 'OpenDuration', -- sum('OpenDuration') has the same
    purpose as with sum('Amount')
    avg(sum('Amount')) over([ .. ] partition by all) as 'global_avg_Amount', --
    avg(sum('Amount')) will give us the average amount of all the opportunities
    in the dataset this number will now be projected on to each record.
    avg(sum('OpenDuration')) over([ .. ] partition by all) as
    'global_avg_OpenDuration'; -- avg(sum('OpenDuration')) will give us the
    average open duration of all the opportunities in the dataset this number
    will now be projected on to each record.

-- Up until now, the previous lines formatted the data so that we can perform
calculations easily. Now we will be performing the actual calculation for the
correlation coefficient. Refer to the formulas used for the correlation
coefficient. The 3 main calculations we will need are going to be covariance,
standard deviation of amount, and standard deviation of open duration.
```



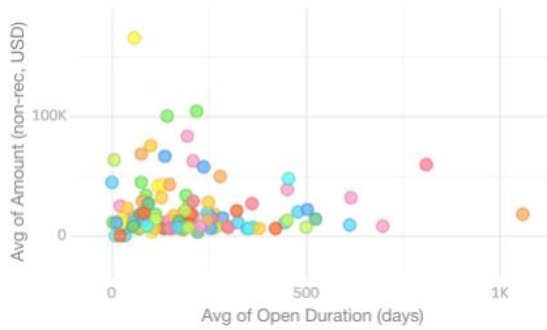
```
-- Because we are going to have to use the (x_i - x_bar) calculation 3 times
-- throughout the formula, we might as well do the calculation once and reuse
-- the value rather than doing the calculation 3 times
q = foreach q generate 'Id',
    'Amount',
    'OpenDuration',
    ('OpenDuration' - 'global_avg_OpenDuration') as 'dur_value', -- (x_i -
-- x_bar), for each opportunity we are calculating the difference between its
-- open duration and the average open duration
    ('Amount' - 'global_avg_Amount') as 'amo_value'; -- (y_i - y_bar), for
-- each opportunity we are calculating the difference between its amount and the
-- average amount

-- Calculating the standard deviation of OpenDuration
dur = group q by all;
dur = foreach dur generate sqrt(power(sum('dur_value'), 2)/count()) as
'stddev';

-- Calculating the standard deviation of Amount
amo = group q by all;
amo = foreach amo generate sqrt(power(sum('amo_value'), 2)/count()) as
'stddev';

-- Calculating the covariance between OpenDuration and Amount
cov = foreach q generate 'dur_value'*'amo_value' as 'cov_value';
cov = group cov by all;
cov = foreach cov generate (sum('cov_value')/count()) as 'cov_value';

-- The correlation coefficient calculation
r = group dur by all, amo by all, cov by all; -- cogroup to bring the 3
-- components of the correlation coefficient together...covariance calculation
-- and the two standard deviations calculations
r = foreach r generate
(sum(cov['cov_value'])/(sum(dur['stddev'])*sum(amo['stddev']))) as
'sum_correlation_coefficient'; -- The sum()'s here do not change the values
-- they are only here because some sort of aggregation is always needed after a
-- group by statement. We could have used max(), min(), or avg() and it would
-- have had the same effect.
```



0.48  
Correlation Coefficient

## Simple Linear Regression

A simple linear regression fits a straight line through a set of points defined on an x-y axis (two measures, one is a predictor and the other is a prediction). It can be expressed as:

$$y = \beta x + \alpha$$

where:

$$\beta = \frac{\text{covariance}_{x,y}}{\text{variance}_x}$$

$$\text{covariance}_{x,y} = \sum_{i=1}^N \frac{(x_i - \bar{x})(y_i - \bar{y})}{N}$$

$$\text{variance}_x = \sum_{i=1}^N \frac{(x_i - \bar{x})^2}{N}$$

$$\alpha = \bar{y} - \beta \bar{x}$$

### Query with comments:

```
q = load "Opportunity";
q = group q by 'Id';
q = foreach q generate 'Id',
    sum('Amount') as 'Amount',
    sum('OpenDuration') as 'OpenDuration',
    avg(sum('Amount')) over([ .. ] partition by all) as 'global_avg_Amount', --
    avg(sum('OpenDuration')) over([ .. ] partition by all) as
    'global_avg_OpenDuration'; --
```

*avg(sum('Amount')) will give us the average amount of all the opportunities in the dataset this number will now be projected on to each record.*

*avg(sum('OpenDuration')) will give us the average open duration of all the opportunities in the dataset this number will now be projected on to each record.*

```
q = foreach q generate 'Id',
```

```

    'Amount',
    'OpenDuration',
    'global_avg_OpenDuration',
    'global_avg_Amount',
    power('OpenDuration' - 'global_avg_OpenDuration', 2) as 'dur_variance', --
    (x_i - x_bar)^2, for each opportunity we are calculating the squared
difference between its open duration and the average open duration
    ('Amount' - 'global_avg_Amount')*(('OpenDuration' -
    'global_avg_OpenDuration')) as 'covariance'; -- (x_i - x_bar)*(y_i - y_bar)

q = group q by 'Id';
q = foreach q generate 'Id',
    sum('Amount') as 'Amount',
    sum('OpenDuration') as 'OpenDuration',
    sum('global_avg_OpenDuration') as 'global_avg_OpenDuration',
    sum('global_avg_Amount') as 'global_avg_Amount',
    ( (sum(sum('covariance')) over([ .. ] partition by all)) /
    (sum(sum('dur_variance')) over([ .. ] partition by all)) ) as 'beta'; --
Calculation to get the beta value of the equation
q = foreach q generate 'Id',
    'Amount',
    'OpenDuration',
    'beta',
    'global_avg_Amount' - ('beta'*'global_avg_OpenDuration') as 'alpha'; --
Calculation to get the alpha value of the equation

-- Getting two points to draw a line
start_point = order q by 'OpenDuration' asc;
start_point = limit start_point 1;
end_point = order q by 'OpenDuration' desc;
end_point = limit end_point 1;
line = union start_point, end_point;
line = foreach line generate 'OpenDuration', ('alpha' + ('beta' *
'OpenDuration')) as 'Predicted_Amount';

```

## Standard Score Normalization

Standard score normalization has many applications. In the context of Wave, it can be used to make comparisons across multiple measures that are not on the same scale. For example, the number of opportunity owners on an account and the dollar amount of those opportunities are on different scales. The amount of opportunities is usually thousands of times larger than the number of owners. Converting the number of opportunities and the amounts to standard scores brings the two numbers to the same scale. This makes it easier to make observations like Account A has fewer opportunity owners than the typical account but is an above average account in terms of dollar amount. A standard score close of zero indicates that the number is

equal to the average. A positive standard score indicates that the number is larger than the average. A negative standard score indicates that the number is smaller than the average.

The formula to get the standard score is the following:

$$\text{standard score} = \frac{x - \bar{x}}{\text{standard deviation}_x}$$

where: 
$$\text{standard deviation}_x = \sqrt{\sum_{i=1}^N \frac{(x_i - \bar{x})^2}{N}}$$

#### Query with comments:

```
q = load "Opportunity";
q = group q by 'Id';
q = foreach q generate 'Id',
    sum('Amount') as 'sum_Amount',
    power(sum('Amount') - (avg(sum('Amount')) over([ .. ] partition by all)),
2) as 'sq_diff'; -- (x_i - x_bar)^2
q = group q by 'Id';
q = foreach q generate 'Id',
    sum('sum_Amount') as 'sum_Amount',
    sum('sum_Amount') - (avg(sum('sum_Amount')) over([ .. ] partition by all))
as 'diff', -- (x_i - x_bar)
    sqrt( (sum(sum('sq_diff')) over([ .. ] partition by all)) /
        (sum(count()) over([ .. ] partition by all)) ) as 'stddev'; --
standard deviation calculation
q = foreach q generate 'Id',
    'sum_Amount',
    ('diff' / 'stddev') as 'z_score';
```

#### Use Case:

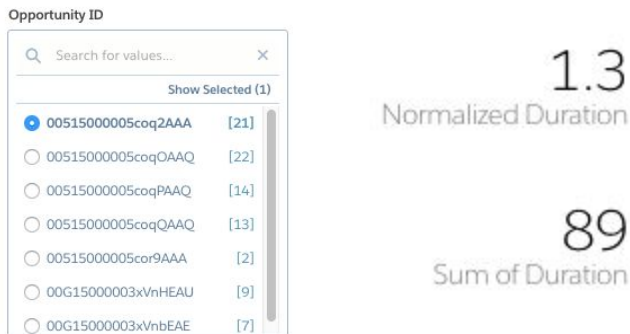
Suppose we have a dashboard KPI for the number of days an opportunity has been open.

Opportunity ID

Opportunity ID	Count
00515000005coq2AAA	[21]
00515000005coq0AAQ	[22]
00515000005coqPAAQ	[14]
00515000005coqQAAQ	[13]
00515000005cor9AAA	[2]
00G15000003xVnHEAU	[9]
00G15000003xVnbEAE	[7]

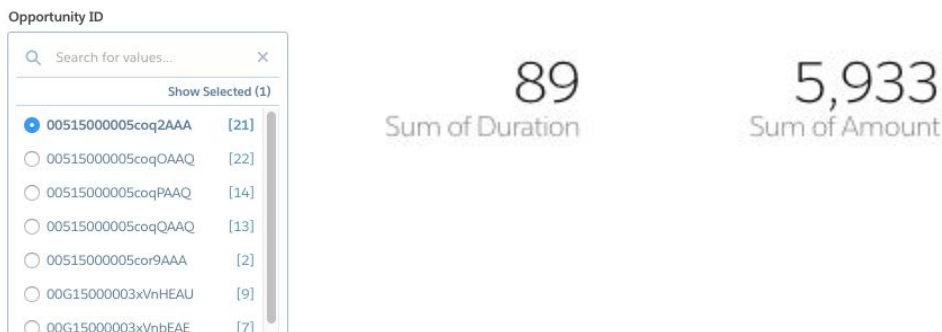
89  
Sum of Duration

Showing that the selected opportunity has been open for 89 days is informative but it could use a little more context. The user may wonder if 89 days is a relatively small or large number, and by itself, this number cannot give us that context.



If we calculate the standard score (1.3), the user can see that 89 days is a longer duration than average. Now this is useful, but we might as well show the average rather than the standard score if this is all we are looking for.

The standard score becomes more valuable when comparing multiple numbers. In this case, the number of days an opportunity has been open and the size of the opportunity.



Again, there's no context for 89 days open and \$5,933 opportunity amount.



The standard scores still give us an understanding of how much smaller or larger the Duration and Amount are compared to their averages, which is information we could have inferred from showing the average. However, now we can also say that, even though the duration and amount of this opportunity are both larger than average, the duration is relatively much larger than the amount. Therefore, the user may want to evaluate if the time spent on this opportunity is worth it.

## Mean and Standard Deviation

Some useful reference lines to transpose onto bar charts using combo charts are the mean and +/- 1 standard deviation.

The formula for the standard deviation is the following:

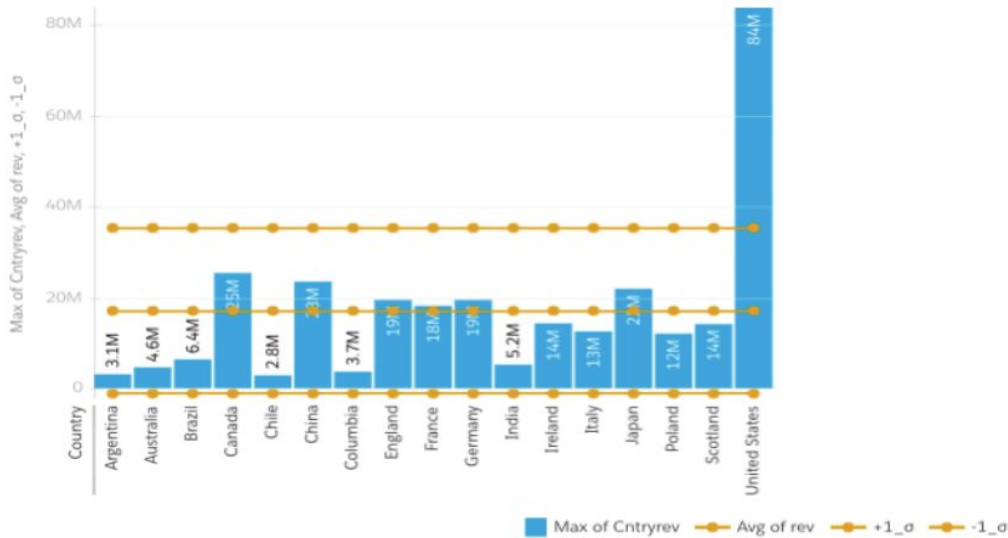
$$standard\ deviation_x = \sqrt{\sum_{i=1}^N \frac{(x_i - \bar{x})^2}{N}}$$

### Query with comments:

```
q = load "ProductSales";
q = group q by 'Country';
-- Get the total revenue sum(sum('Revenue')) by country and get the count
with sum(1)
q = foreach q generate 'Country',
    sum('Revenue') as 'Revenue',
    sum(sum('Revenue')) over ([..] partition by all ) as 'Trevenue',
    sum(1) over ([..] partition by all) as 'Counter';
q = group q by 'Country';
-- Subtract each country total from the average. Average is calculated as
(first('Trevenue') / sum(first('Counter')). Take the absolute of the
difference of country total from average, square it and divide by the number
of countries. The number of countries is sum(first('Counter')). The absolute
can be eliminated because we are squaring anyways.
q = foreach q generate 'Country',
    power(abs(first('Revenue') - (first('Trevenue') / sum(first('Counter'))
over ([..] partition by all))),2)/sum(first('Counter')) over ([..] partition by
all) as 'sqr_diff_Rev',
    first('Trevenue') / sum(first('Counter')) over ([..] partition by all) as
'avg_rev',
    first('Revenue') as 'Cntryrev';
q = group q by 'Country';
-- sqrt(sum(sum('sqr_diff_Rev')) gives the standard deviation for each
country. Here, we are adding 1 standard deviation and subtracting 1 standard
```

deviation from the mean. The reason we are projecting country is because our Bar+Line combo chart needs a dimension and a measure value to draw the line.

```
q = foreach q generate 'Country',
    max('Cntryrev') as 'Cntryrev', round(first('avg_rev')) as 'avg_rev',
    round(first('avg_rev') + sqrt(sum(sum('sqr_diff_Rev')) over ([...] partition
by all))) as 'plus_1_std_dev',
    round(first('avg_rev') - sqrt(sum(sum('sqr_diff_Rev')) over ([...] partition
by all))) as 'minus_1_std_dev';
```



## Quartiles

Quartiles can be used to remove outliers. In general, Quartile Analysis is done by looking at calculating the Lower quartile, Higher quartile and the Interquartile range, to represent the values in the lower ranges, higher ranges and in the middle.

```
|-----|-----|-----|
      Q1       Q3
```

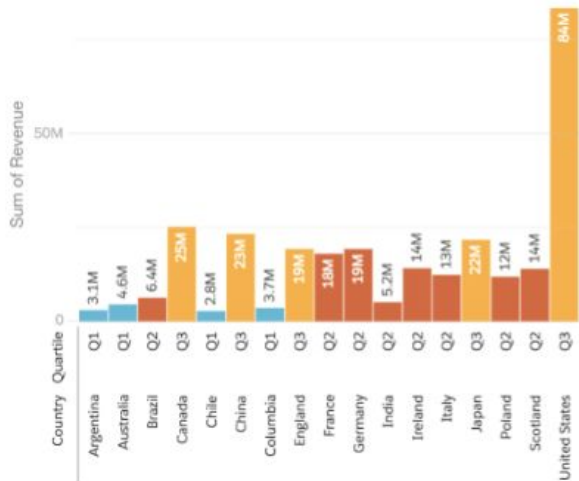
### Query with comments:

```
q = load "ProductSales";
q = group q by 'Country';
-- Getting Total revenue for each country and their rank (calling it Postion)
and ordering by the total revenue
q = foreach q generate 'Country',
    sum('Revenue') as sum_Revenue',
    rank() over ([...] partition by all order by sum('Revenue')) as 'Position',
    sum(1) over ([...] partition by all) as 'count';

-- Q1 represents values at or below position (n+1)/4. Q3 represents values at
or higher than position 3(n+1)/4. The values that fall between Q1 and Q3 are
```

called Q2 which is the Interquartile range. 'count' represents the total number of unique 'countries' in the data.

```
q = foreach q generate 'Country',
    'sum_Revenue',
    ( case
        When 'Position' <= (floor(('count' + 1)/4)) then "Q1"
        when 'Position' >= (floor(((3*'count') + 1)/4)) then "Q3"
        else "Q2" end ) as Quartile;
```



## Percentiles

A few use cases for percentiles are outlier removal/selection, benchmarking, and quantification of data shape (mean and median can sometimes be misleading).

### Query with comments:

```
q = load "Opportunity";
q = group q by 'Owner';
q = foreach q generate 'Owner',
    sum('Amount') as 'sum_Amount',
    row_number() over([ .. ] partition by all order by sum('Amount') asc) as
    'rank_number',
    0.95 * (sum(1) over([ .. ] partition by all)) as 'percentile_value';
q = foreach q generate 'Owner', -- 0.95 specifies which percentile we are
    calculating and we multiple by the size of the row to get which rank
    corresponds to the 95th percentile. Note that this number will not always be
    a round number; we will address this below.
    'sum_Amount',
    'rank_number',
    'percentile_value',
```



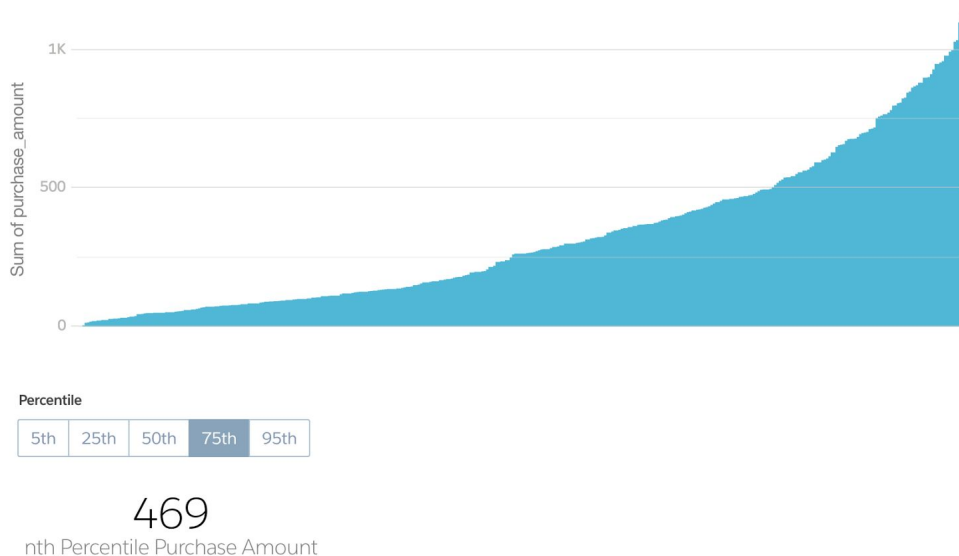
-- The following 3 fields are needed for when the rank of nth percentile is not a round number and falls between two ranks. E.g., rank 3.14 is between rank 3 and rank 4. We will use these three fields to take a weighted average.

```

    floor('percentile_value') as 'rank_floor',
    ceil('percentile_value') as 'rank_ceil',
    'percentile_value' - floor('percentile_value') as 'rank_decimal';
qf = filter q by 'rank_number' == 'rank_floor'; -- Isolate the lower rank.
E.g., rank 3
qc = filter q by 'rank_number' == 'rank_ceil'; -- Isolate the lower rank.
E.g., rank 4
x = group qf by all, qc by all; -- Bring them back together
x = foreach x generate sum(qf['sum_Amount']) + ( qf['rank_decimal'] *
(sum(qf['sum_Amount']) - sum(qc['sum_Amount'])) ) as 'sum_Value'; -- Calculate
the weighted average between those two ranks. If the upper and lower rank are
the same, then this step has no effect but is still executed.

```

The above query calculates the 95th percentile. However, we can easily replace “0.95” with a static step to allow selection of a specific percentile in the dashboard. The 25th percentile is equivalent to the 1st Quartile, 75th to the 3rd Quartile, and 50th to the median.



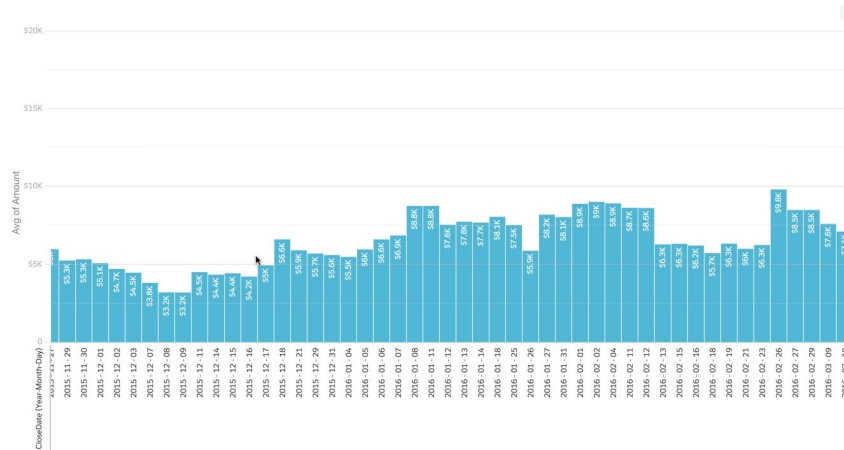
## Windowing Functions

### 14 day moving average

Windowing can be used to compute aggregations over a rolling period. For instance calculating the average value of a measure for the past 14 days.

Query with comments:

```
q = load "Opportunity";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in
["6 years ago".."current year"];
-- For each date
q = group q by ('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day');
-- Calculate the average amount over the last 14 days
q = foreach q generate 'CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day',
avg(sum('Amount')) over([-14 .. 0] partition by all order by ('CloseDate_Year',
'CloseDate_Month', 'CloseDate_Day')) as 'avg_Amount';
q = order q by ('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day');
```



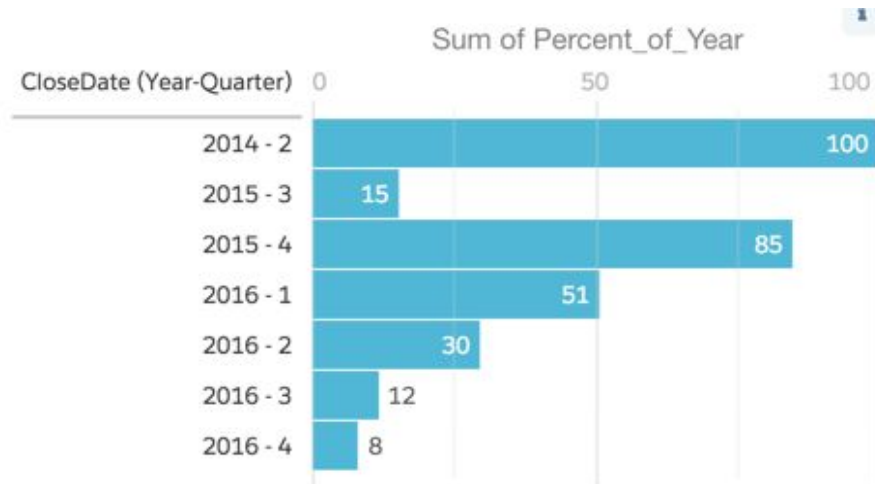
## Percent of year

With windowing, the query to calculate percentages over a period of time can be simplified.

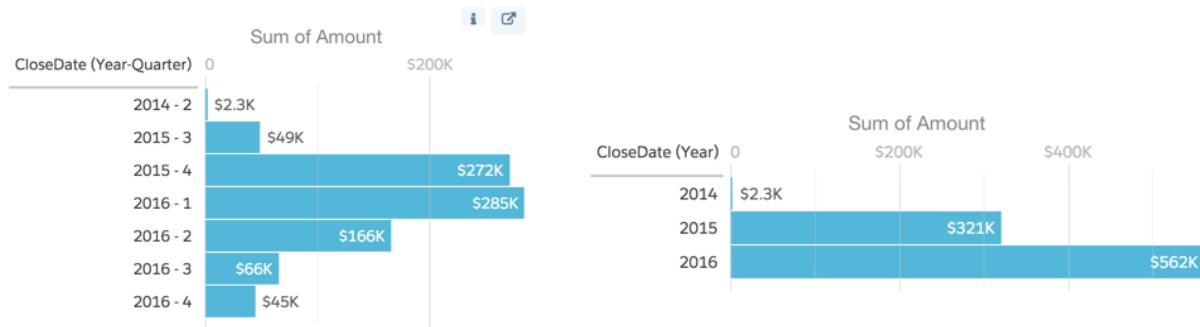
### Query with comments:

```
q = load "Opportunity";
q = filter q by date('CloseDate_Year', 'CloseDate_Month', 'CloseDate_Day') in
["6 years ago".."current year"];
-- For each quarter
q = group q by ('CloseDate_Year', 'CloseDate_Quarter');
-- Calculate the percentage over the total of the same year
q = foreach q generate 'CloseDate_Year' + "~~~" + 'CloseDate_Quarter' as
'CloseDate_Year~~~CloseDate_Quarter',
(sum('Amount') / sum(sum('Amount')) over([...] partition by
'CloseDate_Year' )) as 'sum_Percent_of_Year';
```

### Query Output



For reference Amount by Quarter and Year



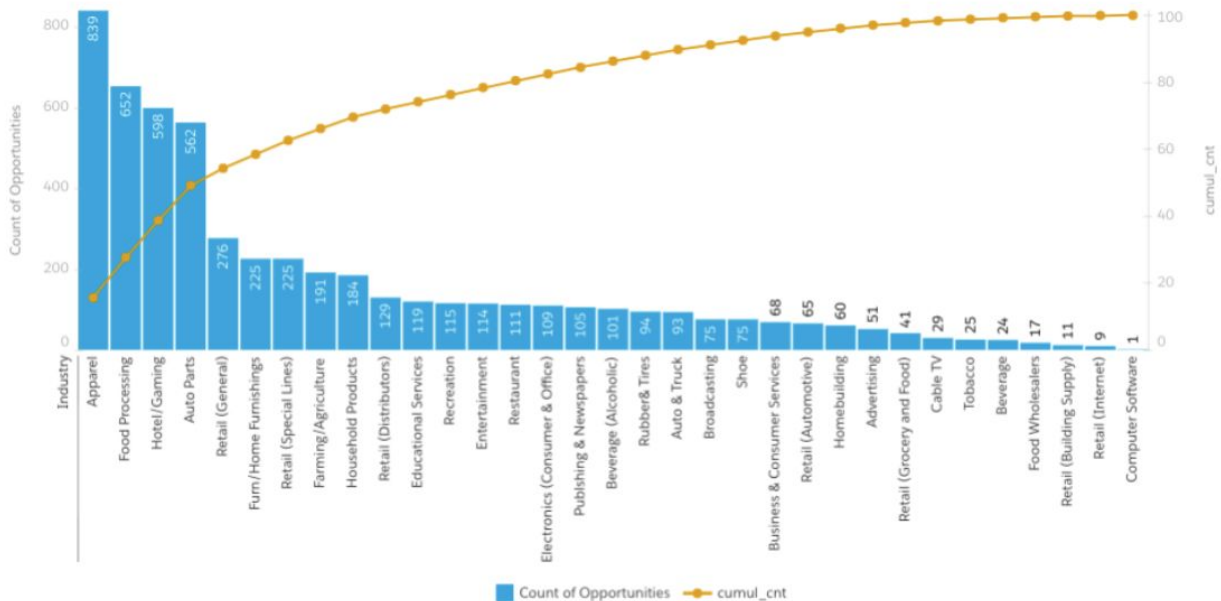
## Pareto

A Pareto chart is useful for identifying which sub group of items makes up x% of the total. The visualization for this is a combo bar-line chart where the bars are the values of individual items sorted in descending order and the line is the cumulative value. For example, a Pareto chart makes it easy to say to that our top 5 accounts make up 80% of our business.

Query with comments:

```
q = load "Sales_new";
q = filter q by 'IsWon' == "FALSE";
q = group q by 'Industry';
-- Calculate number of rows of each Industry as a percentage of total done
by count()/sum(count())
q = foreach q generate 'Industry',
    count() as 'cnt',
    count()/sum(count()) over ([...] partition by all)*100 as 'percent';
q = group q by 'Industry';
-- Calculate a running sum of the percentage share of each industry
q = foreach q generate 'Industry',
    first('cnt') as 'count',
```

```
sum(sum('percent')) over ([..0] partition by all order by first('cnt')
desc) as 'cumul_cnt';
q = order q by 'cumul_cnt';
```



## Top 5 Reps in a Region

If we have to figure out “Top 5 Reps” in each region based on their total contribution to that Region, windowing can be handy.

```
q = load "Opportunity";
q = group q by ('region', 'sales_rep');
q = foreach q generate 'region' as 'Region', 'sales_rep' as 'Sales Rep',
((sum('amount') / sum(sum('amount')) over([..] partition by 'region')) * 100 )
as 'Percent_AmountContribution', rank() over([..] partition by ('region') order
by sum('amount') desc) as 'Rep Rank';
q = filter q by 'Rep Rank' <= 5;
```

`sum('amount')` which will give the Sum for the Sales Rep in that Region and `sum(sum('amount'))` will be the sum for the the entire Region because `over(..)` indicates that aggregate on the entire group which is specified in the Partition and in this case , it is by Region. `Rank()` is to rank the Sales Reps in a region so we can find the top 5.

## Sales Wave Pipeline Trending Customizations

The Pipeline Trending dashboard in Sales Wave is a useful dashboard for looking at your pipeline and seeing how it has changed. We will outline how to make some common customizations to the waterfall chart and table.

## Date Range

The date range filter currently has a static list of date ranges you can use to filter the dashboard with. A common need is to add more date ranges into the filter because you want to look at how your pipeline trended within a particular time frame. You will need to edit the “step\_time” step in order to make these types of changes. New date ranges should be added into the “values”: [ ... ] (note: different from “value”: { ... } ) parameter of “step\_time”.

Dates highlighted in **RED** stay the same and **BLUE** are changed based on the date range you’d like.

### Static Date Range Template:

```
{
  "display": "[start] to [end]",
  "value": {
    "Start_ValidFrom": "dateRange([1, 1, 1], [start - 1 day])",
    "Start_ValidTo": "dateRange([start], [9999, 1, 1])",
    "End_ValidFrom": "dateRange([1, 1, 1], [end])",
    "End_ValidTo": "dateRange([end + 1 day], [9999, 1, 1])",
    "Update_ValidFrom": "dateRange([1, 1, 1], [start - 1 day])",
    "Update_ValidTo": "dateRange([start], [end])",
    "Update_End_ValidFrom": "dateRange([start], [end])",
    "Update_End_ValidTo": "dateRange([end + 1 day], [9999, 1, 1])",
    "CloseDate_1": "dateRange([start], [end])",
    "CloseDate_2": "\"[start]\" .. \"[end]\""
  }
}
```

### Static Date Range Example:

```
{
  "display": "08-12-2015 to 08-19-2015",
  "value": {
    "Start_ValidFrom": "dateRange([1, 1, 1], [2015, 8, 11])",
    "Start_ValidTo": "dateRange([2015, 8, 12], [9999, 1, 1])",
    "End_ValidFrom": "dateRange([1, 1, 1], [2015, 8, 19])",
    "End_ValidTo": "dateRange([2015, 8, 20], [9999, 1, 1])",
    "Update_ValidFrom": "dateRange([1, 1, 1], [2015, 8, 11])",
    "Update_ValidTo": "dateRange([2015, 8, 12], [2015, 8, 19])",
    "Update_End_ValidFrom": "dateRange([2015, 8, 12], [2015, 8, 19])",
    "Update_End_ValidTo": "dateRange([2015, 8, 20], [9999, 1, 1])",
  }
}
```

```

    "CloseDate_1":      "dateRange([2015, 8, 12], [2015, 9, 30])",
    "CloseDate_2":      "\"2015-08-12\"..\"2015-08-19\""
  }
}

```

### Relative Date Range Template:

```

{
  "display": "[start] to [end]",
  "value": {
    "Start_ValidFrom":  "\"current year - 5000 years\"..[start + 1 day]",
    "Start_ValidTo":    "[start]..\"current year + 5000 years\"",
    "End_ValidFrom":    "\"current year - 5000 years\"..[end]",
    "End_ValidTo":      "[end - 1 day]..\"current year + 5000 years\"",
    "Update_ValidFrom": "\"current year - 5000 years\"..[start + 1 day]",
    "Update_ValidTo":   "[start]..[end]",
    "Update_End_ValidFrom": "[start]..[end]",
    "Update_End_ValidTo": "[end - 1 day]..\"current year + 5000 years\"",
    "CloseDate_1":      "[start]..[end]",
    "CloseDate_2":      "[start]..[end]"
  }
}

```

### Relative Date Range Example:

```

{
  "display": "180 to 90 days ago",
  "value": {
    "Start_ValidFrom":  "\"current year - 5000 years\"..\"181 days ago\"",
    "Start_ValidTo":    "\"180 days ago\"..\"current year + 5000 years\"",
    "End_ValidFrom":    "\"current year - 5000 years\"..\"90 days ago\"",
    "End_ValidTo":      "\"89 days ago\"..\"current year + 5000 years\"",
    "Update_ValidFrom": "\"current year - 5000 years\"..\"181 days ago\"",
    "Update_ValidTo":   "\"180 days ago\"..\"90 days ago\"",
    "Update_End_ValidFrom": "\"180 days ago\"..\"90 days ago\"",
    "Update_End_ValidTo": "\"89 days ago\"..\"current year + 5000 years\"",
    "CloseDate_1":      "\"180 days ago\"..\"90 days ago\"",
    "CloseDate_2":      "\"180 days ago\"..\"90 days ago\""
  }
}

```

## Filters

Another common customization is adding more filters to the waterfall chart. You should create filter steps and widgets as you normally would and then manually bind those filters to the “pigql” query in the “waterfall\_chart” step. Also, make sure to apply the same filters to the remaining charts including the waterfall table so that the faceting will be accurate.

### Original query:

```
x1 = load "pipeline_trending";
x1 = filter x1 by 'Opportunity.Owner.Name' in {{
selection(Opportunity_Owner_Name_3) }};
x1 = filter x1 by 'Opportunity.Owner.Role.Name' in {{
selection(Opportunity_Owner_Role_3) }};
x1 = filter x1 by 'Opportunity.Account.Industry' in {{
selection(Account_Industry_12) }};
x1 = filter x1 by 'Opportunity.Account.BillingCountry' in {{
selection(Account_BillingCountry_11) }};
START_REAL = filter x1 by date('ValidFromDate_Year', 'ValidFromDate_Month',
'ValidFromDate_Day') in [{{
no_quote(value(field(selection(step_time), 'Start_ValidFrom')))) }}];
START_REAL = filter START_REAL by date('ValidToDate_Year', 'ValidToDate_Month',
'ValidToDate_Day') in [{{
no_quote(value(field(selection(step_time), 'Start_ValidTo')))) }}];
```

...

```
WATERFALL = union START,upd_full_notinperiod,upd_delta,upd_full_closed, END;
WATERFALL = order WATERFALL by 'Bucket' asc;
```

#### Query with additional filters:

```
x1 = load "pipeline_trending";
x1 = filter x1 by 'Opportunity.Owner.Name' in {{
selection(Opportunity_Owner_Name_3) }};
x1 = filter x1 by 'Opportunity.Owner.Role.Name' in {{
selection(Opportunity_Owner_Role_3) }};
x1 = filter x1 by 'Opportunity.Account.Industry' in {{
selection(Account_Industry_12) }};
x1 = filter x1 by 'Opportunity.Account.BillingCountry' in {{
selection(Account_BillingCountry_11) }};
x1 = filter x1 by 'Opportunity.Account.Market' in {{
selection(Market_Filter_Step) }};
START_REAL = filter x1 by date('ValidFromDate_Year', 'ValidFromDate_Month',
'ValidFromDate_Day') in [{{
no_quote(value(field(selection(step_time), 'Start_ValidFrom')))) }}];
START_REAL = filter START_REAL by date('ValidToDate_Year', 'ValidToDate_Month',
'ValidToDate_Day') in [{{
no_quote(value(field(selection(step_time), 'Start_ValidTo')))) }}];
```

...

```
WATERFALL = union START,upd_full_notinperiod,upd_delta,upd_full_closed, END;
WATERFALL = order WATERFALL by 'Bucket' asc;
```

Here we added a filter for Market into the query. The Market field must first be added to the dataflow. The filter is then added at the end of the x1 stream and is using a selection binding to reference the Market filter step. The rest of the query is the same.

Now, with the new autoFilter feature, you do not have to add any new filters to the x1 stream, you simply set autoFilter to true and new filters that are derived from the same dataset will be automatically added to the step.

An important caveat is that neither adding a filter to the x1 stream nor turning on autoFilter will not work if you are trying to add a filter on Forecast Category. This is because Forecast Category is used throughout the query to determine buckets. It is possible to filter on opportunities that were in particular Forecast Category at the start of the time range (i.e., the Open Pipe (Start) bucket), but the customizations are more complex and will be covered in a separate document dedicated completely for the Sales Wave Pipeline Trending dashboard.

## Relabeling Buckets

Relabeling buckets is an easy customization to make through the dataset XMD. We recommend that you do relabeling in the XMD as opposed to the SAQL query if it is only the label name that you want to change and your logic behind the buckets stays the same.

XMD:

```
{
  "labels": {
    "keys": {
      "Bucket": {
        "0_START": "Open Pipe (Start)",
        "1_NEW": "New",
        "2_REOPEN": "Reopen",
        "3_EXPAND": "Expand",
        "4_MOVED_IN": "Moved In",
        "5_MOVED_OUT": "Moved Out",
        "6_REDUCE": "Reduce",
        "7_CLOSED_WON": "Closed Won",
        "8_CLOSED_LOST": "Closed Lost",
        "9_END": "Open Pipe (Today)"
      },
      ...
    }
  },
  ...
}
```

If you wanted to relabel “Closed Lost” as “Lost”, you would make that edit in the XMD. For more information on XMD see the [XMD Reference Page](#).



## Other Customizations

Some other customizations that we see are with the Buckets, Stacked Waterfall, and Forecast Category Filters. These customizations will be detailed in a separate document focused specifically on the queries in the Pipeline Trending dashboard.

## What's New

### Version 2.0

- Updated dashboard JSON syntax
- Windowing Use Cases
  - Windowing Alternative to Period over Period Calculations
  - Windowing Alternative to MoM on most recent available month
  - 14 day moving average
  - Percent of year
  - Pareto
  - Top 5 reps in a region
- Math
  - Correlation Coefficient
  - Simple Linear Regression
  - Standard Score Normalization
  - Mean and Standard Deviation
  - Quartiles
  - Percentiles
- Miscellaneous
  - Field-to-Field Filters
  - Co-Group as Filter
  - Timeline Comparisons
- Sales Wave Pipeline Trending Customizations
  - Date Range
  - Filters
  - Relabeling

### Version 2.1

- Updated best practice for null handling in results bindings
- Removed incorrect comment that order by statements can only use measures
- Removed incorrect comment about effects of order by clause in windowing functions
- Optimized queries
- Workaround for count() windowing function, sum(1)

- Added order by *statement* after windowing functions that require ordering. The order by *clause* in windowing functions does not order the projection; it only orders how the data is calculated.
- autoFilter can be used for Sales Wave waterfall chart filtering
- Updated broken links

## Credits

*Sathish Sadagopan*, [Principal Solution Engineer](#), For providing examples and documenting new SAQL use cases.