Analytics Bindings Developer Guide

Salesforce, Summer '22

@salesforcedocs

Last updated: June 30, 2022

CONTENTS

INTERACTIONS IN CRM ANALYTICS DASHBOARD

Interactions allow you to work with different components in a dashboard. You control the interactions by binding queries to each other. There are two types of interactions: selection interaction and results interaction. The selection or results of one query triggers updates in other queries in the dashboard.

Note: Prior to the Spring '20 release, interactions were called bindings. Prior to the Winter '20 release, queries were called steps.

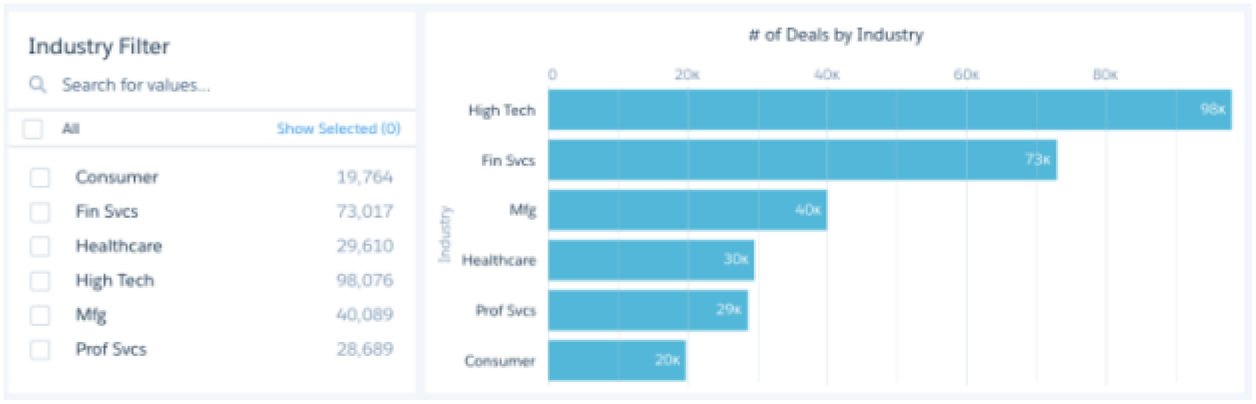
Tip: Before you create interactions to make widgets interactive, consider faceting. Facets are the simplest and most common way to specify interactions between widgets. When faceted, selections made in one widget automatically filter all other widgets using queries from the same dataset. Faceting is easy to set up, but it is limited. It can only filter other queries and works only on queries from the same dataset. To create interactions outside this scope, use interactions.

For more information about queries, see Widget Steps in a CRM Analytics Dashboard. For more information about faceting, see Making Widgets Interactive Using Facets and Bindings.

Selection Interaction

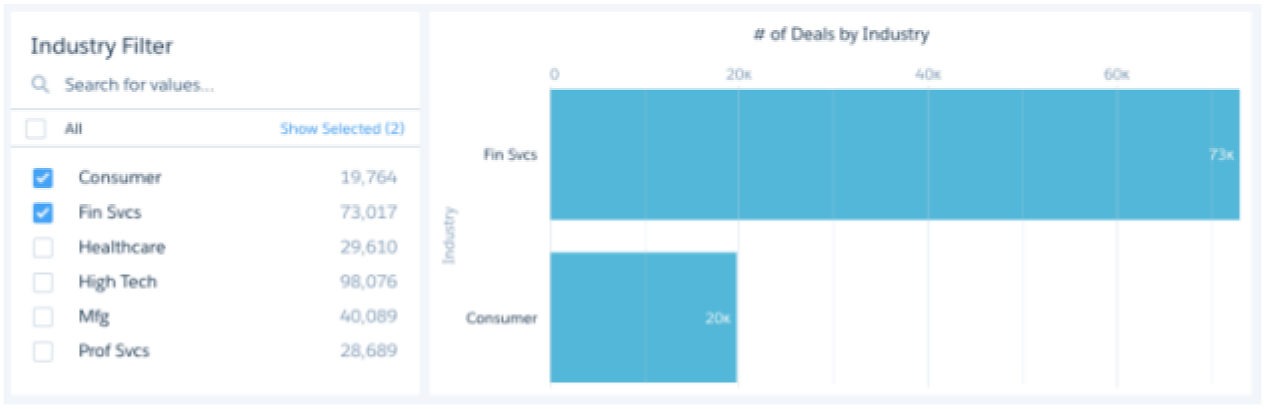
Selection interaction is a method used to update a query based on the selection in another query. Selection interactions are evaluated each time the user selects something in a widget.

For example, you have the following dashboard.



Consumer	19,764
Fin Svcs	73,017
Healthcare	29,610
High Tech	98,076
Mfg	40,089
Prof Svcs	28,689

A selection in the Industry Filter widget filters the results in the # of Deals by Industry chart.



✓	Consumer	19,764
✓	Fin Svcs	73,017
	Healthcare	29,610
	High Tech	98,076
	Mfg	40,089
	Prof Svcs	28,689

A selection interaction can be used to:

- Specify interactions between widgets which use queries from different datasets.
- In addition to filters, specify the measures, groupings, and other aspects of a query.
- Set widget display properties for some widget types (number and chart only).

Note: Interactions on widget titles and subtitles are ignored when you open the widget in an explorer lens.

Result Interaction

Results interaction is a method used to update a query based on the results of another query.

A results interaction is typically used to:

- Define intermediate results for a complex calculation. For example, to calculate the total opportunity amount for the top-five products, use one query to calculate the top-five products. Then use those results to filter another query that calculates the total number of open cases for each product.
- Set an initial filter selection for a dashboard based on a characteristic of the logged-in user, like their country.
- Dynamically change the display of a widget based on the results of a query. For example, you can configure a number widget to show different colors based on the value of the measure. (In dashboard designer only.)

Interactions in Dashboard Designer Dashboard

The dashboard designer treats selection and results interactions the same. Both types of interactions operate on tabular data and return complete rows, even for columns not used in widgets. The dashboard designer treats multiple selections as tabular data and single selections as a single row of tabular data. The designer expresses each row of tabular data in the form of an array of objects, where each object is keyed by the column name.

Syntax

You must specify the right syntax when creating an interaction in a dashboard designer dashboard. The syntax is different for each dashboard designer.

To specify a selection or results interaction in the dashboard designer, use the following syntax.

```
<stepID>.<result|selection>
```

For example,

```
mySourceStep.selection
```

Note: When specifying the query, you specify the ID, not the label.

Interaction Functions

Dashboard designer dashboards support a variety of interaction functions that get data from a source query, manipulate it, and serialize it to make it consumable by the target query.

If the input data has empty results, the function returns a null value. If the specified location of the data doesn’t exist, an error occurs. For example, the interaction is defined to get data from row 3, but row 3 doesn’t exist. An error also occurs if the shape of the input data doesn’t meet the requirement for the function.

Interaction functions operate on input data with one the following shapes.

- Scalar value, like 0, "this is scalar", or null.
- One-dimensional array, like ([1, 2, 3]) or (["one","two","three"])

- Two-dimensional array (an array of arrays), like ([[1, 2], [3, 4]])

The required shape of the input data varies with each function. After processing the data, the functions can change the shape of the data.

Each interaction consists of nested functions. Each interaction must have one data selection function and one data serialization function. Optionally, interactions can have multiple data manipulation functions. (The following sections describe these types of functions.) The following example illustrates how nested functions in an interaction work together to produce the expected result for a target query in which they are defined. The example is based on the following interaction.

```
coalesce(cell(mySourceStep.selection, 0, \"grouping\"), \"state\").asString()
```

The `mySourceStep` query has the following input data.

display	grouping
Regional Area	region
Country	country
State	state

Technically, the data for this query is stored as a two-dimensional array, where each row is stored as a map of key-value pairs.

```
[
    ["display":"Regional Area", "grouping":"region"],
    ["display":"Country", "grouping":"country"],
    ["display":"State", "grouping":"state"]
]
```

At runtime, CRM Analytics evaluates the interaction functions, starting with the innermost function. Using that logic, CRM Analytics evaluates the example's interaction functions in the following order.

Function	Description

`mySourceStep.selection`	CRM Analytics returns each row of selected data as a map of key-value pairs. If a single selection is made, only one row returns. If multiple selections are made, multiple rows return.
	`[`
	`    ["display":"Regional Area",`
	`"grouping":"region"],`
	`    ["display":"Country",`
	`"grouping":"country"],`
	`    ["display":"State", "grouping":"state"]`
	`]`

`cell``(mySourceStep.selection, 0, \"grouping\")`	The `cell` function returns a scalar value from the `"grouping"` column of the first row (indicated by the 0 index) returned by `mySourceStep.selection`. Based on the selection, the return value can be `"region"`, `"country"`, or `"state"`. If no selection is made, the function returns `null`.

Function	Description
`coalesce``(cell(mySourceStep.selection, 0, \"grouping\"), \"state\")`	The `coalesce` function returns the value of the `cell` function if a selection was made. If no selection was made, the `coalesce` function returns the specified default value `"state"`. Tip: Use the `coalesce` function to provide a default value so that an error doesn't occur when a null value is encountered.
`coalesce(cell(mySourceStep.selection, 0, \"grouping\"), \"state\")`**`.asString()`**	The `.asString` function returns the result of the `coalesce` function as a SAQL string.

To see how these functions are used in interactions, see Use Cases.

Data Selection Functions

A data selection function selects data from a source. The source can be either a selection or results of a query. The function returns a table of data, where each column has a name, and each row has an index, starting with 0. From the table, you can select one or more rows, one or more columns, or a cell to include in your interaction.

Data Manipulation Functions

A data manipulation function changes the data into the format required by the data serialization function (see the next section). You can apply a manipulation function on the results from a data selection or other data manipulation function. If the input data is null, the manipulation function returns null, unless otherwise specified.

Data Serialization Functions

Serialization functions convert the data into the form expected by the query in which the interaction is inserted. For example, if the interaction is used in a compact-form query, use the `asObject()` function to format the data into a one-dimensional object.

Data Serialization Functions for SQL and SOQL

SQL and SOQL serialization functions convert the data into the form expected by the query in which the interaction is inserted.

Data Selection Functions

A data selection function selects data from a source. The source can be either a selection or results of a query. The function returns a table of data, where each column has a name, and each row has an index, starting with 0. From the table, you can select one or more rows, one or more columns, or a cell to include in your interaction.

In cases where multiple rows or columns of data are selected, the function returns a two-dimensional array. When a single row or single column is selected, the function returns a one-dimensional array. When a cell is selected, the function returns a scalar value. The function returns null if the source is empty. If the function tries to select data that doesn't exist, an interaction error occurs. For example, the table only has two rows, but you try to select data from the third row.

cell Function

Returns a single cell of data as a scalar, like "This salesperson rocks", 2, or `null`. An error occurs if the `rowIndex` is not an integer, the `columnName` is not a string, or the cell doesn't exist in the table.

column Function

Returns one column of data (as a one-dimensional array) or multiple columns of data (as a two-dimensional array).

row Function

Returns one row of data (as a one-dimensional array) or multiple rows (as a two-dimensional array). For selection interactions, you typically use this function to return the first row or all rows. For results interactions, you might want specific rows. To determine the row index, display the query results in a values table.

cell Function

Returns a single cell of data as a scalar, like "This salesperson rocks", 2, or `null`. An error occurs if the `rowIndex` is not an integer, the `columnName` is not a string, or the cell doesn't exist in the table.

Syntax

```
cell(source, rowIndex, columnName)
```

Arguments

Argument	Description
source	(Required) Specify the name of the query and `selection` or `result`.
rowIndex	(Required) Specify the row using its index. Row index starts at 0.
columnName	(Required) Specify the column name.

The following example is based on the `myStep` source query. Assume that `myStep.selection` retrieves the following rows from the query.

```
[
    {stateName: 'CA', Amount:100},
    {stateName: 'TX', Amount:200},
    {stateName: 'OR'', Amount:300},
    {stateName: 'AL', Amount:400},
]
```

Although CRM Analytics doesn't store this data as a table, let's show the data in this format to make it easier to understand the example.

(row index)	stateName	Amount
0	CA	100
1	TX	200
2	OR	300
3	AL	400

Example:

```
cell(myStep.selection, 1, "stateName")
```

Output:

```
"TX"
```

column Function

Returns one column of data (as a one-dimensional array) or multiple columns of data (as a two-dimensional array).

Syntax

```
column(source, [columnNames...])
```

Arguments

Argument	Description
source	(Required) Specify the name of the query and `selection` or `result`.
columnNames	(Required) Specify an array of column names. The order of the listed columns affects the order that the columns appear in the output. The order is important for serialization functions. For example, the `asOrder` function requires the first element to be a field name and the second to be the direction.

The following examples are based on the `myStep` source query. Assume that `myQuery.selection` retrieves the following rows from the query.

```
[
    {stateName: ‘CA’, Amount:100},
    {stateName: ‘TX’, Amount:200},
    {stateName: ‘OR’’, Amount:300},
    {stateName: ‘AL’, Amount:400},
]
```

Although CRM Analytics doesn’t store this data as a table, let’s show the data in this format to make it easier to understand the examples that follow.

(row index)	stateName	Amount
0	CA	100
1	TX	200
2	OR	300
3	AL	400

Example:

```
column(myStep.selection, ["stateName"])
```

Output:

```
["CA", "TX", "OR", "AL"]
```

Example:

```
column(myStep.selection, [])
```

Output:

```
[ ["CA", "TX", "OR", "AL"], ["100", "200″," "300", "400"] ]
```

row Function

Returns one row of data (as a one-dimensional array) or multiple rows (as a two-dimensional array). For selection interactions, you typically use this function to return the first row or all rows. For results interactions, you might want specific rows. To determine the row index, display the query results in a values table.

Syntax

```
row(source), [rowIndices...], [columnNames...])
```

Arguments

Argument	Description
source	(Required) Specify the name of the query and `selection` or `result`.
rowIndices	(Required) Specify an array of row indices, where each element of the array identifies a row. Row index `0` identifies the first row. To include all rows, specify an empty array.
columnNames	(Optional) Specify an array of column names to select and order them. If not specified, all columns are selected and every row has the same order of columns. However, that order isn't guaranteed to be the same across different queries.

The following examples are based on the `myStep` source query. Assume that `myStep.selection` retrieves the following rows from the query.

```
[
    {stateName: ‘CA’, Amount:100},
    {stateName: ‘TX’, Amount:200},
    {stateName: ‘OR’’, Amount:300},
    {stateName: ‘AL’, Amount:400},
]
```

Although CRM Analytics doesn't store this data as a table, let's show the data in this format to make it easier to understand the examples that follow.

(row index)	stateName	Amount
0	CA	100
1	TX	200
2	OR	300
3	AL	400

Example:

```
row(myStep.selection, [0], ["Amount"])
```

Output:

```
["100"]
```

Example:

```
row(myStep.selection, [0,2], [])
```

Output:

```
[ ["CA", "100"], ["OR", "300"] ]
```

Example:

```
row(myStep.selection, [], ["stateName"])
```

Output:

```
[ ["CA"], ["TX"], ["OR"], ["AL"] ]
```

Example:

```
row(myStep.selection, [0,2], ["stateName", "Amount"])
```

Output:

```
[ ["CA", "100"], ["OR", "300"] ]
```

Data Manipulation Functions

A data manipulation function changes the data into the format required by the data serialization function (see the next section). You can apply a manipulation function on the results from a data selection or other data manipulation function. If the input data is null, the manipulation function returns null, unless otherwise specified.

Note: If data manipulation isn't required, add a data serialization function to the results of the data selection functions.

coalesce Function
Returns the first non-null source from a list of sources. Useful for providing a default value in case a function returns a null value.

concat Function
Joins streams from multiple sources into a one- or two-dimensional array. Null sources are skipped.

flatten Function
Flattens a two-dimensional array into a one-dimensional array.

join Function
Converts a one- or two-dimensional array into a string by joining the elements using the specified token. An error occurs if the data has any other shape.

slice Function
Selects one or more values from a one-dimensional array given a start and, optionally, an end position, and returns a one-dimensional array. An error occurs if the start value is greater than the end value. Negative indices are supported.

toArray Function
Converts scalars to a one-dimensional array, and one-dimensional arrays to a two-dimensional array. For example, use this function to convert the scalar result of a cell function to an array, which is required by compact-form measures, groups, and order clauses. The function returns an error if the input is a series of static one-dimensional arrays, a two-dimensional array, or a mix of scalars and one-dimensional arrays.

valueAt Function
Returns the single scalar value at the given index.

coalesce Function

Returns the first non-null source from a list of sources. Useful for providing a default value in case a function returns a null value.

Syntax

```
coalesce(source1, source2,...)
```

Arguments

Argument	Description
source	(Required) Source can be the results of a data selection or other data manipulation function. The source can have any shape.

Example:

```
coalesce(cell(step1.selection, 0, "column1"), "green")
```

Output: The output is the result returned by `cell(step1.selection, 0, "column1")`. However, if `cell(step1.selection, 0, "column1")` returns `null`, then the output is

```
"green"
```

For an application of this function in an interactions use case, see Change the Map Type Based on a Toggle Widget.

concat Function

Joins streams from multiple sources into a one- or two-dimensional array. Null sources are skipped.

Syntax

```
concat(source1, source2,...)
```

Arguments

Argument	Description
source	(Required) Each source can be the results of a data selection or other data manipulation function. Each source must be either a one- or two-dimensional array. An error occurs if you try to concatenate data from sources of different shapes. For example, the following function produces an error: `concat(["a", "b"], ["c", "d", "e"])`.

Example:

```
concat(["a", "b"], ["c", "d"])
```

Output:

```
["a", "b", "c", "d"]
```

Example:

```
concat([["a", "b"]], [["c", "d"]])
```

Output:

```
[["a", "b"], ["c", "d"]]
```

flatten Function

Flattens a two-dimensional array into a one-dimensional array.

Syntax

```
flatten(source)
```

Arguments

Argument	Description
source	(Required) Source can be the results of a data selection or other data manipulation function. The source must be a two-dimensional

Argument	Description
	array; otherwise, an error occurs because there's no reason to flatten a one-dimensional array or scalar.

Example:

```
flatten([["CDG", "SAN"], ["BLR", "HND"], ["SMF", "JFK"])
```

Output:

```
["CDG", "SAN", "BLR", "HND", "SMF", "JFK"]
```

join Function

Converts a one- or two-dimensional array into a string by joining the elements using the specified token. An error occurs if the data has any other shape.

Syntax

```
join(source, token)
```

Arguments

Argument	Description
source	(Required) Source can be the results of a data selection or other data manipulation function. The source must be a two-dimensional array; otherwise, an error occurs.
token	(Required) Any string value, like + or ,.

Example:

```
join(["a", "b", "c"], "+")
```

Output:

```
["a+b+c"]
```

Example:

```
join([["a", "b", "c"], [1, 2]], "~")
```

Output:

```
["a~b~c~1~2"]
```

slice Function

Selects one or more values from a one-dimensional array given a start and, optionally, an end position, and returns a one-dimensional array. An error occurs if the start value is greater than the end value. Negative indices are supported.

Syntax

```
slice(source, start, end)
```

Arguments

Argument	Description
source	(Required) Source can be the results of a data selection or other data manipulation function. The source can have any shape.
start	(Required) Index that identifies the start value in the array. For example, 0 represents the first element in the array.
end	(Optional) Index that identifies the end value in the array.

Example:

```
slice(step.selection, -1, 0)
```

Returns the last selected row.

toArray Function

Converts scalars to a one-dimensional array, and one-dimensional arrays to a two-dimensional array. For example, use this function to convert the scalar result of a cell function to an array, which is required by compact-form measures, groups, and order clauses. The function returns an error if the input is a series of static one-dimensional arrays, a two-dimensional array, or a mix of scalars and one-dimensional arrays.

Syntax

```
toArray (source1, source2,...)
```

Argument

Argument	Description
source	(Required) The input must be scalars, including static values, or one-dimensional arrays.

Example: Consider the following Opportunities query result.

Oppty_Name	Owner	Region	Amount	Created_Date
Alpha	Danny	Americas	1000	2/20/2017
Bravo	Danny	Americas	500	4/15/2017
Charlie	Jeff	EMEA	2000	5/1/2017

These interactions use the Opportunities query as a source.

Interaction	Result
`toArray( cell(Opportunities.selection, 0, "Region") )`	["Americas"]
`toArray( "APAC" )`	["APAC"]
`toArray( cell(Opportunities.selection, 0, "Region"), "APAC" )`	["Americas", "APAC"]
`toArray( column(Opportunities.selection, ["Oppty_Name"]), column(Opportunities.selection, ["Region] )`	[["Alpha", "Bravo", "Charlie"], ["Americas", "Americas", "EMEA"]]
`toArray( column(Opportunities.selection, ["Oppty_Name", "Region"]) )`	Error. The column interaction function returns a two-dimensional array, which is an invalid input.
`toArray( [1, 2, 3], [4, 5, 6] )`	Error. The input can't be a series of static one-dimensional arrays.

valueAt Function

Returns the single scalar value at the given index.

Syntax

```
valueAt(source, index)
```

Arguments

Argument	Description
source	(Required) Source can be the results of a data selection or other data manipulation function. The source can have any shape.
index	(Required) Negative indexes are supported. If you specify an index that doesn't exist, the function returns null.

Example:

```
valueAt(cell(step.selection, 0, "column"), -1)
```

Returns the last selected value.

Data Serialization Functions

Serialization functions convert the data into the form expected by the query in which the interaction is inserted. For example, if the interaction is used in a compact-form query, use the `asObject()` function to format the data into a one-dimensional object.

asDateRange() Function
: Returns the date range filter condition as a string for a SAQL query. The date range is inclusive. Use the string as part of a filter based on dates.

asEquality() Function
: Returns an equality or "in" filter condition as a string for a SAQL query. The input data must be a scalar, one-dimensional array, or two-dimensional array.

asGrouping() Function
: Returns a grouping as a string. Can also return multiple groupings

asObject() Function
: Passes data through with no serialization. Returns data as an object (an array of strings).

asOrder() Function
: Returns the sort order as a string for a SAQL query.

asProjection() Function
: Returns the query expression and alias as a string that you can use to project a field in a query. The query expression determines the value of the field. The alias is the field label.

asRange() Function
: Returns a range filter condition as a string for a SAQL query. The range is inclusive.

asString() Function
: Serializes a scalar, one-dimensional array, or two-dimensional array as a string. Escapes double quotes in strings.

asDateRange() Function

Returns the date range filter condition as a string for a SAQL query. The date range is inclusive. Use the string as part of a filter based on dates.

The input data must be a one- or two-dimensional array. If the input data is a one-dimensional array with two elements, the function uses the first element as the minimum and the second element as the maximum. Null results in `fieldName in all`, which applies no filter.

Syntax

```
<input data>.asDateRange(fieldName)
```

Arguments

Argument	Description
fieldName	(Required) The name of the date field.

The following example is based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
[
    {min: 1016504910000, max: 1281655993000}
]
```

Example:

```
row(stepFoo.selection, [0], ["min", "max"]).asDateRange("date(year, month, day)")
```

Output:

```
date(year, month, day) in [dateRange([2002,3,19], [2010,8,12])]
```

See also Date Range Filters.

asEquality() Function

Returns an equality or "in" filter condition as a string for a SAQL query. The input data must be a scalar, one-dimensional array, or two-dimensional array.

If a single field name is provided, the returned string contains the `in` operator for a one-dimensional array (`fieldName in ["foo", "bar"]`) or the equality operator for a scalar (`fieldName == "foo"`).

If multiple field names are provided, the returned string contains a composite filter. For this case, a two-dimensional array is expected. The number of values in each array must match the number of specified fields.

In Spring '21, the behavior of null values has changed due to the addition of nulls in group values. The updated behavior if the input to this function is `null`, is the function now returns `<fieldName> is null`, instead of `<fieldName> by all`, where `<fieldName>` is the first field. For example, if `cell(step1.selection, 0, "column1")` evaluates to `null`, `cell(step1.selection, 0, "column1").asEquality("field1")` now evaluates to `'field1' is null`, instead of `'field1' by all`. If there are no selected rows, `cell(step1.selection, "column1")`, the function evaluates to `<fieldName> in all`.

Syntax

```
<input data>.asEquality(fieldName)
```

Arguments

Argument	Description
fieldName	(Required) The name of the field.

The following examples are based on the `myStep` source query. Assume that `myStep.selection` is the following.

```
[
    {grouping: "first", measure: 19}
    {grouping: "second", measure: 32}
]
```

Example:

```
cell(myStep.selection, 1, "measure").asEquality("bar")
```

Output:

```
bar == 32
```

Example:

```
column(myStep.selection, ["grouping"]).asEquality("bar")
```

Output:

```
bar in ['first','second']
```

Example: This example illustrates how a lack of input data is handled. Imagine that `myStep.selection` resolves as

```
[]
```

```
cell(myStep.selection, 0, "value").asEquality("bar")
```

Output:

```
'bar' in all
```

Example: The next examples illustrate how null values and empty arrays are evaluated when present in the `myStep.selection`.

```
[
    {grouping: "first", measure: null},
    {grouping: "second", measure: 32}
]
```

```
cell(myStep.selection, 0, "measure").asEquality("bar")
```

Output:

```
'bar' is null
```

```
[
    {label: "Bananas", value: []}
]
```

```
cell(myStep.selection, 0, "value").asEquality("bar")
```

Output

```
'bar' in all
```

See also Filters.

asGrouping() Function

Returns a grouping as a string. Can also return multiple groupings

The input data must be a scalar or one-dimensional array of groupings. Null results in a `group by all`.

Syntax

```
<input data>.asGrouping()
```

Let's look at some examples where the selection determines the groupings in a SAQL-form query. The following examples are based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
[
    {grouping: "first", alias: "foo"}
    {grouping: "second", alias: "bar"}
]
```

Example:

```
cell(stepFoo.selection, 1, "grouping").asGrouping()
```

Output:

```
'second'
```

Example: To make the interaction return multiple fields for the grouping, replace the `cell` interaction function with a `column` function and update the arguments.

```
column(stepFoo.selection, ["grouping"]).asGrouping()
```

Output:

```
('first', 'second')
```

See also Group Interactions.

asObject() Function

Passes data through with no serialization. Returns data as an object (an array of strings).

Syntax

```
<input data>.asObject()
```

Example:

```
column(StaticMeasureNamesStep.selection, [\"value\"]).asObject()
```

For an application of this function in an interactions use case, see Bind Parts of a Query.

Example:

```
cell(static_1.selection, 0, \"value\").asObject()
```

asOrder() Function

Returns the sort order as a string for a SAQL query.

The input data must be a scalar, one-dimensional array, or two-dimensional array. A two-dimensional array is treated as a tuple of interactions.

Syntax

```
<input data>.asOrder()
```

The following example is based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
[
    {order: "first", direction: "desc"}
    {order: "second", direction: "asc"}
]
```

Example:

```
cell(stepFoo.selection, 1, "order").asOrder()
```

Output:

```
'second'
```

Example:

```
column(stepFoo.selection, ["order"]).asOrder()
```

Output:

```
('first', 'second')
```

Example:

```
row(stepFoo.selection, [], ["order", "direction"]).asOrder()
```

Output:

```
('first' desc, 'second' asc)
```

See also Order Interactions.

asProjection() Function

Returns the query expression and alias as a string that you can use to project a field in a query. The query expression determines the value of the field. The alias is the field label.

Use this function to write a `foreach` statement for a field projection. The function concatenates the query expression, 'as', and the field label in the following format.

```
<query_expression> as <field_label>
```

Here's a sample projection that rounds the price to two decimals and stores the result in the SalesPrice field.

```
q = foreach q generate round(Price, 2) as SalesPrice;
```

In this example, `round(Price, 2)` is the expression and `SalesPrice` is the field label.

Syntax

```
<input data>.asProjection()
```

The following example is based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
[
    {expression: "first", alias: "foo"}
    {expression: "second", alias: "bar"}
]
```

Example:

```
stepFoo.selection, [0], ["expression", "alias"]).asProjection()
```

Output:

```
first as 'foo'
```

Example:

```
stepFoo.selection, [], ["expression", "alias"]).asProjection()
```

Output:

```
first as 'foo', second as 'bar'
```

See also Projection Interactions.

asRange() Function

Returns a range filter condition as a string for a SAQL query. The range is inclusive.

The input data must be a one-dimensional array with at least two elements. The function uses the first as the minimum and the second as the maximum. null results in fieldName by all, which applies no filter.

Syntax

```
<input data>.asRange(fieldName)
```

Arguments

Argument	Description
fieldName	(Required) The name of the field.

The following example is based on the `myStep` source query. Assume that `myStep.selection` retrieves the following rows from the query.

```
[
    {grouping: "first", measure: 19}
    {grouping: "second", measure: 32}
]
```

Example:

```
row(myStep.selection, [0], ["min", "max"]).asRange("bar")
```

Output:

```
bar >= 19 && bar <= 32
```

See also Range Filters.

asString() Function

Serializes a scalar, one-dimensional array, or two-dimensional array as a string. Escapes double quotes in strings.

Syntax

```
<input data>.asString()
```

Example:

```
cell(stepOpportunity.selection, 1, "measure").asString()
```

Example:

```
cell(color_1.result, 0, "color").asString()
```

For an application of this function in an interactions use case, see Highlight Values with Color Coding.

Data Serialization Functions for SQL and SOQL

SQL and SOQL serialization functions convert the data into the form expected by the query in which the interaction is inserted.

asSQLGrouping() Function
Returns a grouping as a string in a SQL or SOQL query. Can also return multiple groupings.

asSQLHaving() Function

Returns the grouped results for a SQL or SOQL query restricted by a conditional filter logic applied on aggregate functions. This function is similar to the `asSQLWHERE` function.

asSQLOrder() Function

Returns the sort order as a string for a SQL or SOQL query.

asSQLSelect() Function

Returns the SQL or SOQL query expression and alias as a string that you can use to project a field in the query. The query expression determines the value of the field. The alias is the field label.

asSQLWhere() Function

Returns the SQL or SOQL query results restricted by a conditional filter logic applied on a measure, dimension, or date.

asSQLGrouping() Function

Returns a grouping as a string in a SQL or SOQL query. Can also return multiple groupings.

The input data must be a scalar or one-dimensional array of groupings. Null or no results will return an ungrouped query.

Syntax

```
<input data>.asSQLGrouping(SQLType, includeKeywords)
```

Arguments

Argument	Description
SQLType	(Required) Specify SQL or SOQL variant type. Valid values are: • `dataset` • `snowflake` • `sobject` Note: For SOQL, use `sobject`.
includeKeywords	(Optional) The default value is `false`.

Let's look at some examples where the selection determines the groupings in a SQL and SOQL-form query. The following examples are based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
[
    {grouping: "first", alias: "foo"}
    {grouping: "second", alias: "bar"}
]
```

Example:

```
(cell(stepFoo.selection, 0, "grouping").asSQLGrouping("snowflake", true))
```

Output in standard dataset or Snowflake live dataset:

```
GROUP BY "first"
```

```
(cell(stepFoo.selection, 0, "grouping").asSQLGrouping("sobject", true))
```

Output in SOQL:

```
GROUP BY first
```

Example:

```
(cell(stepFoo.selection, 0, "grouping").asSQLGrouping("snowflake", false))
```

Output in standard dataset or Snowflake live dataset:

```
"first"
```

```
(cell(stepFoo.selection, 0, "grouping").asSQLGrouping("sobject", false))
```

Output in SOQL:

```
first
```

Example: To make the interaction return multiple fields for the grouping, replace the `cell` interaction function with a `column` function and update the arguments.

```
(column(stepFoo.selection, ["alias"]).asSQLGrouping("snowflake", true))
```

Output in standard dataset or Snowflake live dataset:

```
GROUP BY "foo", "bar"
```

```
(column(stepFoo.selection, ["alias"]).asSQLGrouping("sobject", true))
```

Output in SOQL:

```
GROUP BY foo, bar
```

```
(column(stepFoo.selection, ["alias"]).asSQLGrouping("snowflake", false))
```

Output in standard dataset or Snowflake live dataset:

```
"foo", "bar"
```

```
(column(stepFoo.selection, ["alias"]).asSQLGrouping("sobject", false))
```

Output in SOQL:

```
foo, bar
```

asSQLHaving() Function

Returns the grouped results for a SQL or SOQL query restricted by a conditional filter logic applied on aggregate functions. This function is similar to the `asSQLWHERE` function.

Syntax

```
<input data>.asSQLHaving(SQLType, includeKeywords)
```

Argument	Description
SQLType	(Required) Specify SQL or SOQL variant type. Valid values are: • `dataset` • `snowflake` • `sobject` Note: For SOQL, use `sobject`.
includeKeywords	(Optional) The default value is `false`.

The following examples are based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query. This function is similar to the `asSQLWhere` function.

Note: The >< and >=<= operators aren't supported.

Example:

```
(toArray("SUM(Revenue)", ">",column(stepFoo.selection,
["Revenue"]).asSQLHaving("snowflake", true))
```

Output in standard or Snowflake live dataset:

```
HAVING "SUM(Revenue)" > 10
```

```
(toArray("SUM(Revenue)", ">",column(stepFoo.selection,
["Revenue"]).asSQLHaving("sobject", true))
```

Output in SOQL:

```
HAVING SUM(Revenue) > 10
```

Example:

```
(toArray("SUM(Revenue)", ">",column(stepFoo.selection,
["Revenue"]).asSQLHaving("dataset"))
```

Output in standard or Snowflake live dataset:

```
"SUM(Revenue)" > 10
```

Output in SOQL:

```
SUM(Revenue) > 10
```

Example:

```
([].asSQLHaving('dataset'))
or ([].asSQLHaving('dataset', 'false'))
```

Output in standard or Snowflake live dataset:

```
""
```

Output in Snowflake live dataset:

```
""
```

Output in SOQL:

```
""
```

asSQLOrder() Function

Returns the sort order as a string for a SQL or SOQL query.

The input data must be a scalar, one-dimensional array, or two-dimensional array. A two-dimensional array is treated as a tuple of interactions.

Syntax

```
<input data>.asSQLOrder(SQLType, includeKeywords)
```

Arguments

Argument	Description
SQLType	(Required) Specify SQL or SOQL variant type. Valid values are: • `dataset` • `snowflake` • `sobject` Note: For SOQL, use `sobject`.
includeKeywords	(Optional) The default value is `false`.

The following example is based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
values:[
    {order: "first", direction: "desc"}
    {order: "second", direction: "asc"}
]
```

Example:

```
(cell(stepFoo.selection, 0, "order").asSQLOrder("snowflake", true))
```

Output in standard dataset or Snowflake live dataset:

```
ORDER BY "first"
```

```
(cell(stepFoo.selection, 0, "order").asSQLOrder("sobject", true))
```

Output in SOQL:

```
ORDER BY first
```

Example:

```
column(stepFoo.selection, ["order"]).asSQLOrder("snowflake"))
```

Output in standard dataset or Snowflake live dataset:

```
"first", "second"
```

```
column(stepFoo.selection, ["order"]).asSQLOrder("sobject"))
```

Output in SOQL:

```
first, second
```

Example:

```
(row(stepFoo.selection, [], ["order", "direction"]).asSQLOrder("snowflake", true))
```

Output in standard dataset or Snowflake live dataset:

```
ORDER BY "first" DESC, "second", ASC
```

```
(row(stepFoo.selection, [], ["order", "direction"]).asSQLOrder("sobject", true))
```

Output in SOQL:

```
ORDER BY first DESC, second ASC
```

asSQLSelect() Function

Returns the SQL or SOQL query expression and alias as a string that you can use to project a field in the query. The query expression determines the value of the field. The alias is the field label.

Syntax

```
<input data>.asSQLSelect(SQLType)
```

Argument	Description
SQLType	(Required) Specify SQL or SOQL variant type. Valid values are: • `dataset` • `snowflake` • `sobject` Note: For SOQL, use `sobject`.

The following example is based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

```
values:[
    {expression: "SUM("first")", alias: "foo"}
    {expression: "second", alias: "bar"}
]
```

Example:

```
(row(stepFoo.selection, [0], ["expression", "alias"]).asSQLSelect("snowflake"))
```

Output in standard dataset or Snowflake live dataset:

```
SUM("first") AS "foo"
```

```
(row(stepFoo.selection, [0], ["expression", "alias"]).asSQLSelect("sobject"))
```

Output in SOQL:

```
SUM("first") foo
```

Example:

```
(row(stepFoo.selection, [0], ["expression"]).asSQLSelect("snowflake"))
```

Output in standard dataset or Snowflake live dataset:

```
SUM("first")
```

```
(row(stepFoo.selection, [0], ["expression"]).asSQLSelect("sobject"))
```

Output in SOQL:

```
SUM("first")
```

Example:

```
(row(stepFoo.selection, [], ["expression", "alias"]).asSQLSelect("snowflake"))
```

Output in standard dataset or Snowflake live dataset:

```
SUM("first") AS "foo",  "second" AS "bar"
```

```
(row(stepFoo.selection, [], ["expression", "alias"]).asSQLSelect("sobject"))
```

Output in SOQL:

```
SUM("first") foo, second bar
```

asSQLWhere() Function

Returns the SQL or SOQL query results restricted by a conditional filter logic applied on a measure, dimension, or date.

Syntax

```
<input data>.asSQLWhere(SQLType, includeKeywords)
```

Argument	Description
SQLType	(Required) Specify SQL or SOQL variant type. Valid values are: • `dataset` • `snowflake` • `sobject` Note: For SOQL, use `sobject`.
includeKeywords	(Optional) The default value is `false`.

The following examples are based on the `stepFoo` source query. Assume that `stepFoo.selection` retrieves the following rows from the query.

Example:

```
([].asSQLWhere('dataset'))
or ([].asSQLWhere('dataset', 'false'))
```

Output in standard dataset:

```
""
```

Output in Snowflake live dataset:

```
""
```

Output in SOQL:

```
""
```

Example:

```
((toArray("foo", "IN", column(stepFoo.selection, ["foo"])).asSQLWhere("snowflake",
true))
toArray("foo", "IN", column(stepFoo.selection, ["foo"])  =>  ["foo", "IN", ["a", "b"]]
```

Output in standard dataset:

```
WHERE "foo" IN ('a', 'b')
```

Output in Snowflake live dataset:

```
WHERE "foo" IN ('a', 'b')
```

```
((toArray("foo", "IN", column(stepFoo.selection, ["foo"])).asSQLWhere("sobject", true))

toArray("foo", "IN", column(stepFoo.selection, ["foo"])  =>  ["foo", "IN", ["a", "b"]]
```

Output in SOQL:

```
WHERE foo IN ('a', 'b')
```

Example:

```
(toArray("bar", "><", column(stepFoo.selection, ["bar"])).asSQLWhere("snowflake",
false))
toArray("bar", "><", column(stepFoo.selection, ["bar"])  =>  ["bar", "><", [10, 100]]
```

Output in standard dataset:

```
"bar" > 10 AND "bar" < 100
```

Output in Snowflake live dataset:

```
"bar" > 10 AND "bar" < 100
```

```
(toArray("bar", "><", column(stepFoo.selection, ["bar"])).asSQLWhere("sobject", false))

toArray("bar", "><", column(stepFoo.selection, ["bar"])  =>  ["bar", "><", [10, 100]]
```

Output in SOQL:

```
bar > 10 AND bar < 100
```

Example: (Multiple filters)

```
(toArray(toArray("foo", "IN", column(stepFoo.selection, ["foo"]),
toArray("bar", "><", column(stepFoo.selection, ["bar"]))).asSQLWhere("snowflake",
true))
```

Output in standard dataset or Snowflake live dataset:

```
WHERE "foo" IN ('a', 'b') AND ("bar" > 10 AND "bar" < 100)
```

```
(toArray(toArray("foo", "IN", column(stepFoo.selection, ["foo"]),
toArray("bar", "><", column(stepFoo.selection, ["bar"]))).asSQLWhere("sobject", true))
```

Output in SOQL:

```
WHERE foo IN ('a', 'b') AND (bar > 10 AND bar < 100)
```

Example:

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("dataset", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [1016504910000, 1281655993000]]
```

Output in standard dataset:

```
WHERE ("CreatedDate"  >= DATE '2002-3-19' AND "CreatedDate" <= '2010-8-12'))
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("snowflake", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [1016504910000, 1281655993000]]
```

Output in Snowflake live dataset:

```
WHERE ("CreatedDate" BETWEEN to_date('2002-3-19') AND to_date('2010-8-12'))
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("sobject", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [1016504910000, 1281655993000]]
```

Output in SOQL:

```
WHERE (CreatedDate >= 2002-03-19 AND CreatedDate <= 2010-08-12)
```

Example:

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("dataset"))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [null, 1281655993000]]
```

Output in standard dataset:

```
"CreatedDate" <= DATE '2010-8-12'
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("snowflake"))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [null, 1281655993000]]
```

Output in Snowflake live dataset:

```
 "CreatedDate" <= to_date('2010-8-12')
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("sobject"))
```

```
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [null, 1281655993000]]
```

Output in SOQL:

```
CreatedDate <= 2010-08-12
```

Example:

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("dataset", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [2015, 8, 1], [2021, 3, 1]]
```

Output in standard dataset:

```
WHERE ("CreatedDate" >= DATE '2015-8-1' AND CreatedDate <= '2021-3-1'))
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("snowflake", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [2015, 8, 1], [2021, 3, 1]]
```

Output in Snowflake live dataset:

```
WHERE ("CreatedDate" BETWEEN to_date('2015-8-1') AND to_date('2021-3-1'))
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("sobject", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", [2015, 8, 1], [2021, 3, 1]]
```

Output in SOQL:

```
WHERE (CreatedDate >= 2015-08-01 AND CreatedDate <= 2021-03-01))
```

Example:

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("snowflake", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", ["quarter", -2], ["quarter", 3]
```

Output in Snowflake live dataset:

```
WHERE ("CreatedDate" >= date_trunc(quarter, dateadd(quarter, -2, current_date()))AND
"CreatedDate" <= date_trunc(quarter, dateadd(quarter, 3, current_date())))
```

```
(toArray("CreatedDate", "BETWEEN", column(stepFoo.selection,
["CreatedDate"])).asSQLWhere("sobject", true))
toArray("CreatedDate", "BETWEEN", column(stepFoo.selection, ["CreatedDate"])) =>
["CreatedDate", "BETWEEN", ["quarter", -2], ["quarter", 3]
```

Output in SOQL:

```
WHERE (CreatedDate >= LAST_N_QUARTERS:2 AND CreatedDate <= NEXT_N_QUARTERS:3)
```

Interaction Limitations for Dashboard Designer Dashboards

CRM Analytics supports interactions in numerous places, but not everywhere. Review the following limitations before you create any interactions.

You can bind widget properties in chart and number widgets only. You can bind all properties, except the following ones.

Widget Type	Unsupported Widget Properties
Number	• borderEdges • borderWidth • compact • exploreLink • measureField • textAlignment
Chart	• borderEdges • borderRadius • borderWidth • exploreLink • measureField

You can't bind the following query properties:

Query Property
`selectMode`

Interaction Errors

If you create an invalid interaction in a dashboard, the widget that uses the interaction displays an error in the dashboard.

Generally, there are two types of errors.

Validation errors

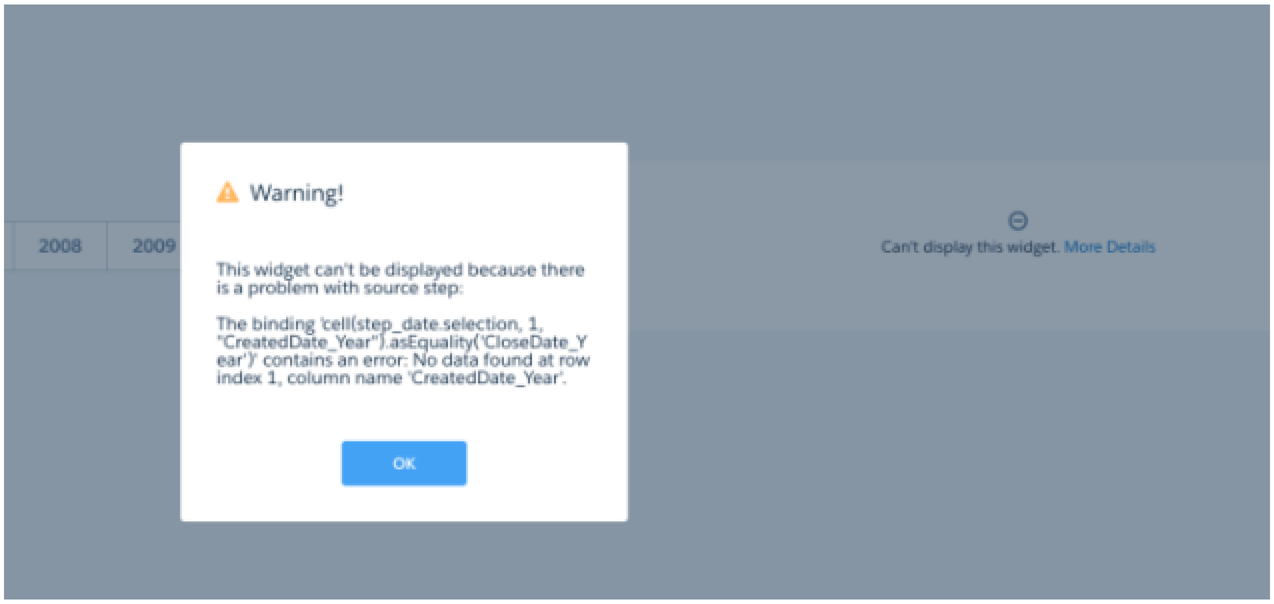
These errors occur when CRM Analytics is unable to parse the interaction due to the wrong syntax or illegal arguments used in your interactions. Another typical issue is that you didn't escape double quotes when they are inside other double quotes. For example, notice how the inner set of double quotes is escaped.

```
"numberColor": "{{cell(color_1.result, 0, \"color\").asString()}}"
```

Execution errors

These errors occur when CRM Analytics executes the interaction and either the expected columns or rows are missing or the data is in the wrong shape. For example, the interaction received a row, when it expected a cell.

Review error messages to understand how to resolve interaction issues. For example, here's an example of an interaction error in a dashboard.



This widget can't be displayed because there is a problem with source step:

The binding 'cell(step_date.selection, 1, "CreatedDate_Year").asEquality('CloseDate_Year')' contains an error: No data found at row index 1, column name 'CreatedDate_Year'.

Use Cases

To help you better understand how to build interactions, look at these different interaction use cases.

To get hands-on with interactions, check out the Einstein Analytics Learning Adventure app available on Salesforce AppExchange. The app walks you through best-practice interaction examples for building powerful, dynamic apps. Download the app today and get your hands dirty!

Bind Parts of a Query

You can dynamically set parts of a query based on the selection or results of another query. For example, you can set the grouping in a query based on the grouping selected in a chart.

Bind Queries from Different Datasets

You can bind queries from different datasets. For example, the following dashboard contains two charts, each based on its own dataset.

Bind a Custom Query with Other Queries

You can create custom queries to specify your own values for a query, instead of getting values from a query. For example, you might create a custom query to show "Top 5 Customers" and "Bottom 5 Customers" in a toggle widget. After you create the ustom query, to make it interact with the other widgets in the dashboard, manually bind the custom query to the queries of the other widgets.

Bind Widget Properties

In a dashboard designer dashboard only, you can implement interactions to dynamically change properties for number or chart widgets.

Bind the Initial Filter Selection

You can use a results interaction to dynamically set the initial selection of a query based on a characteristic of the logged-in user. For example, you can filter a region-based dashboard based on the country of the logged-in user.

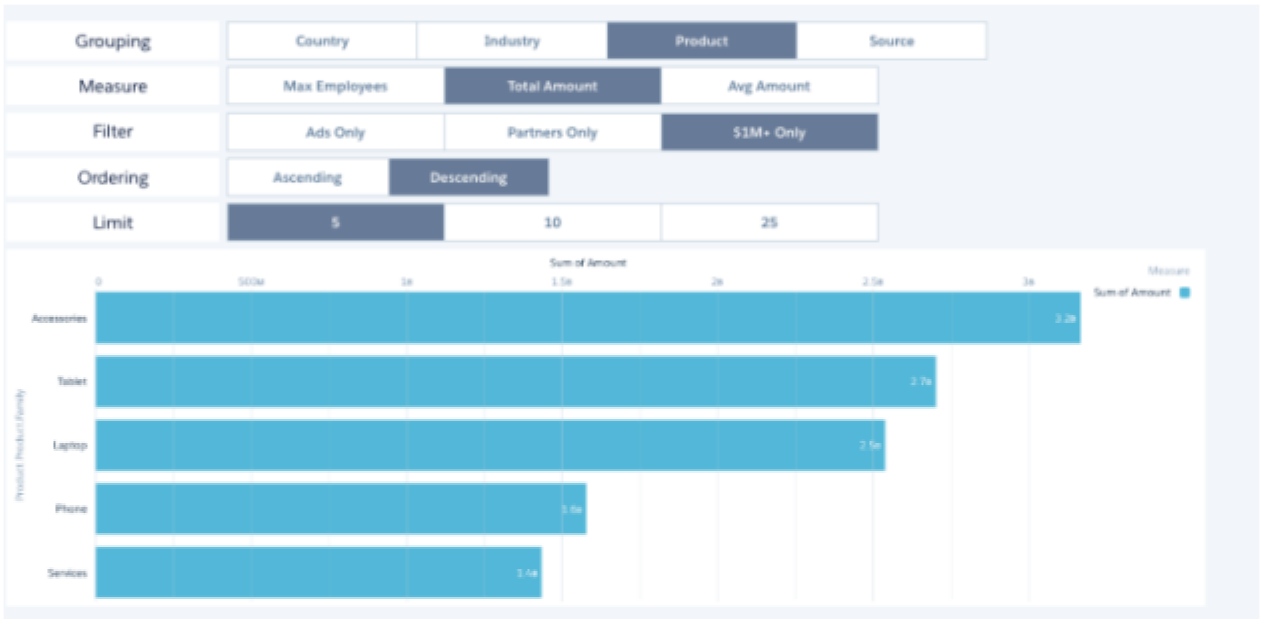
Create Deeper Dependencies with Nested Interactions

Nested interactions enable you to create deeper dependencies among widgets.

Bind Parts of a Query

You can dynamically set parts of a query based on the selection or results of another query. For example, you can set the grouping in a query based on the grouping selected in a chart.

Before we discuss how to bind the different parts of the query, let's look at a comprehensive example. This example illustrates what the interactions look like for different parts of a query. The chart is bound based on selections for grouping, measure, filter, order, and limit. When you make a selection in one of the toggle widgets, the chart morphs to visualize the results of the modified query.



Here's the JSON for the queries that power this dashboard. The Account_BillingCount_1 query is the underlying query for the chart widget. This query contains multiple interactions based on other queries.

```
"steps": {
            "Account_BillingCount_1": {
                "datasets": [
                    {
                        "id": "0FbB00000000oEkKAI",
                        "label": "Opportunities",
                        "name": "opportunity",
                        "url": "/services/data/v38.0/wave/datasets/0FbB00000000oEkKAI"
                    }
                ],
                "isFacet": true,
                "isGlobal": false,
                "query": {
                    "measures": "{{column(StaticMeasureNames.selection,
[\"value\"]).asObject()}}",
                   "limit": "{{column(StaticLimits.selection, [\"value\"]).asObject()}}",

                    "groups": "{{column(StaticGroupingNames.selection,
[\"value\"]).asObject()}}",
                  "filters": "{{column(StaticFilters.selection, [\"value\"]).asObject()}}",

                   "order": "{{column(StaticOrdering.selection, [\"value\"]).asObject()}}"

                },
                "selectMode": "single",
                "type": "aggregateflex",
                "useGlobal": true,
                "visualizationParameters": {
                    "visualizationType": "hbar",
                    "options": {}
                }
            },
            "StaticGroupingNames": {
                "datasets": [],
                "dimensions": [],
                "isFacet": true,
                "isGlobal": false,
                "selectMode": "single",
                "start": {
                    "display": [
                        "Country"
                    ]
                },
                "type": "staticflex",
                "useGlobal": true,
                "values": [
                    {
                        "display": "Country",
                        "value": "Account.BillingCountry"
                    },
                    {
                        "display": "Industry",
                        "value": "Account.Industry"
                    },
                    {
                        "display": "Product",
                        "value": "Product.Product.Family"
                    },
                    {
                        "display": "Source",
                        "value": "Account.AccountSource"
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "StaticFilters": {
                "datasets": [],
                "dimensions": [],
                "isFacet": true,
                "isGlobal": false,
                "selectMode": "single",
                "start": {
                    "display": "Ads Only"
                },
                "type": "staticflex",
                "useGlobal": true,
                "values": [
                    {
                        "display": "Ads Only",
                        "value": [
                            "LeadSource",
                            [
                                "Advertisement"
                            ],
                            "in"
                        ]
                    },
                    {
                        "display": "Partners Only",
                        "value": [
                            "Account.Type",
                            [
                                "Partner"
                            ],
                            "in"
                        ]
                    },
                    {
                        "display": "$1M+ Only",
                        "value": [
                            "Amount",
                            [
                                [
                                    1000000,
                                    11921896
                                ]
                            ],
                            ">=<="
                        ]
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "StaticOrdering": {
                "datasets": [],
                "dimensions": [],
                "isFacet": true,
                "isGlobal": false,
                "selectMode": "single",
                "start": {
                    "display": "Ads Only"
                },
                "type": "staticflex",
                "useGlobal": true,
                "values": [
                    {
                        "display": "Ascending",
                        "value": [
                            -1,
                            {
                                "ascending": true
                            }
                        ]
                    },
                    {
                        "display": "Descending",
                        "value": [
                            -1,
                            {
                                "ascending": false
                            }
                        ]
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "StaticLimits": {
                "datasets": [],
                "dimensions": [],
                "isFacet": true,
                "isGlobal": false,
                "selectMode": "single",
                "start": {
                    "display": [
                        "5"
                    ]
                },
                "type": "staticflex",
                "useGlobal": true,
                "values": [
                    {
                        "display": "5",
                        "value": 5
                    },
                    {
                        "display": "10",
                        "value": 10
                    },
                    {
                        "display": "25",
                        "value": 25
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "StaticMeasureNames": {
                "datasets": [],
                "dimensions": [],
                "isFacet": true,
                "isGlobal": false,
                "selectMode": "singlerequired",
                "start": {
                    "display": [
                        "Total Amount"
                    ]
                },
                "type": "staticflex",
                "useGlobal": true,
                "values": [
                    {
                        "display": "Max Employees",
                        "value": [
                            "max",
                            "Account.NumberOfEmployees"
                        ]
                    },
                    {
                        "display": "Total Amount",
                        "value": [
                            "sum",
                            "Amount"
                        ]
                    },
                    {
                        "display": "Avg Amount",
                        "value": [
                            "avg",
                            "Amount"
                        ]
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "Account_AccountSourc_1": {
                "datasets": [
                    {
                        "id": "0FbB00000000oEkKAI",
                        "label": "Opportunities",
                        "name": "opportunity",
                        "url": "/services/data/v38.0/wave/datasets/0FbB00000000oEkKAI"
                    }
                ],
                "isFacet": true,
                "isGlobal": false,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ],
                    "groups": [
                        "Account.AccountSource"
                    ],
                    "order": [
                        [
                            -1,
                            {
                                "ascending": false
                            }
                        ]
                    ]
                },
                "type": "aggregateflex",
                "useGlobal": true,
                "visualizationParameters": {
                    "visualizationType": "hbar",
                    "options": {}
                }
            }
        },
        "widgetStyle": {
            "backgroundColor": "#FFFFFF",
            "borderColor": "#E6ECF2",
            "borderEdges": [],
            "borderRadius": 0,
            "borderWidth": 1
        }
```

Note: If you bind a measure or grouping in a query used for a chart created during or after Spring '18, you must also replace the `columnMap` section in the widget-level chart JSON with an empty `columns` array. For more information, see Measure Interactions and Group Interactions.

Measure Interactions

Bind the measure to allow the dashboard viewer to select which measures to show in a widget. For example, you can show different measures in a chart based on the selection in a toggle widget.

Filter Interactions

You can create different types of filters in a SAQL query. The following sections walk you through some example filters that use different types of interactions.

Projection Interactions

Use the `asProjection()` serialization function to specify the projection of a field in a SAQL query.

Group Interactions
Bind the grouping to allow the dashboard viewer to select which dimensions to group the results by. For example, you can show different groupings in a chart based on the selection in a toggle widget.

Order Interactions
Use the `asOrder()` serialization function to specify the sort order in a SAQL query.

Limit and Offset Interactions
You can also bind the limit and offset of a SAQL query. These interactions don't require data serialization functions.

Measure and Group Bindings in Compact-Form and SAQL-Form Queries
Bindings can be used both in compact-form queries and SAQL-form queries.

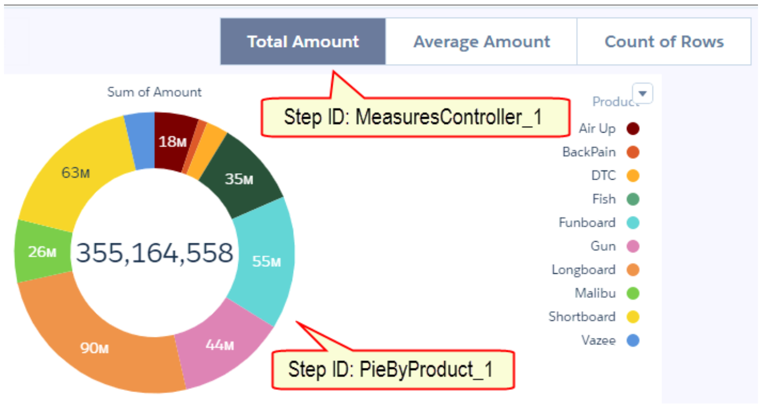
Measure Interactions

Bind the measure to allow the dashboard viewer to select which measures to show in a widget. For example, you can show different measures in a chart based on the selection in a toggle widget.

To dynamically set the measure in a query based on a selection, complete the following tasks.

- Bind the `measures` property of the query.
- If the query is used for a chart created during or after Spring '18, replace the `columnMap` section of the widget with an empty `columns` array. Why? Because when you change the query, the set of fields will likely be different from what's in the `columnMap` section. When you replace the `columnMap` property with an empty `columns` array, the system remaps the columns based on the new query definition.

Let's look at an example where we bind the measure for a donut chart based on the selection in the toggle widget.



The toggle widget uses the following custom query.

```
"MeasuresController_1": {
    "type": "staticflex",
    "label": "MeasuresController",
    "values": [
        {
            "display": "Total Amount",
            "step_property": [ "sum", "Amount" ]
        },
        {
            "display": "Average Amount",
            "step_property": [ "avg", "Amount" ]
        },
        {
            "display": "Count of Rows",
            "step_property": [ "count", "*" ]
        }
    ],
    "selectMode": "singlerequired",
    "start": {
        "display": [ "Total Amount" ]
    },
    "broadcastFacet": true,
    "groups": [],
    "numbers": [],
    "strings": []
}
```

Each toggle option has one display label (`display`) that appears in the toggle. It also has one value (`step_property`) that determines the measure.

Let's bind the `step_property` field of the custom query (`MeasuresController_1`) to the measure in the donut chart's step (`PieByProduct_2`). Any selection in the custom query passes the aggregation method (like sum or count) and the measure field to the `PieByProduct_2` query.

```
"PieByProduct_2": {
    "label": "PieByProduct_2",
    "query": {
        "measures": [
            "{{ cell(MeasuresController_1.selection, 0, \"step_property\").asObject() }}"

        ],
        "groups": [ "Product" ]
    },
    "visualizationParameters": {

        ...

    },
    "receiveFacet": true,
    "selectMode": "single",
    "type": "aggregateflex",
    "isGlobal": false,
    "useGlobal": true,
    "broadcastFacet": true,
    "datasets": [
        {
            "id": "0FbB00000000q5gKAA",
            "label": "Flexy Sales",
            "name": "Flexy_Sales",
            "url": "/services/data/v42.0/wave/datasets/0FbB00000000q5gKAA"
        }
    ]
}
```

When you create the donut chart, by default, the widget (`chart_2`) contains the `columnMap` section that maps measures and groupings to chart attributes.

```
"chart_2": {
    "type": "chart",
    "parameters": {
        "visualizationType": "pie",
        "step": "PieByProduct_2",
        "columnMap": {
            "trellis": [],
            "dimension": [ "Product" ],
            "plots": [ "sum_Amount" ]
        },

        ...

        }
    }
}
```

Note: The properties under the `columnMap` property vary based on the chart type.

To enable the interaction to work, replace the `columnMap` section with an empty `columns` array because interactions cannot be used to specify `columnMap`.

```
"chart_2": {
    "type": "chart",
    "parameters": {
        "visualizationType": "pie",
        "step": "PieByProduct_2",
        "columns" : [],

        ...

        }
    }
}
```

Filter Interactions

You can create different types of filters in a SAQL query. The following sections walk you through some example filters that use different types of interactions.

Filters
You can bind filters based on certain conditions. CRM Analytics supports multiple operators that provide flexibility when defining the conditions.

Range Filters
Use the `asRange()` serialization function to bind filters based on numeric ranges.

Date Range Filters
Use the `asDateRange()` serialization function to bind filters based on date ranges. You can create filters using absolute or relative date ranges.

Filters

You can bind filters based on certain conditions. CRM Analytics supports multiple operators that provide flexibility when defining the conditions.

Filter Example (SAQL Form)

Let's say you have the following results from the source query.

```
[
    {grouping: "first", measure: 19}
    {grouping: "second", measure: 32}
]
```

You can bind a filter using the `asEquality()` interaction function. The following filter condition determines whether the returned value equals "bar."

```
q = filter q by {{cell(stepFoo.selection, 1, "measure").asEquality("bar")}};
```

After evaluating the interaction based on the data returned from the source query, CRM Analytics produces the following filter.

```
q = filter q by bar == 32;
```

Note: If a selection returns multiple values, `asEquality()` inserts the 'in' operator, instead of ==, in the filter statement. For example, the following filter condition determines if any value in the "grouping" column equals "bar."

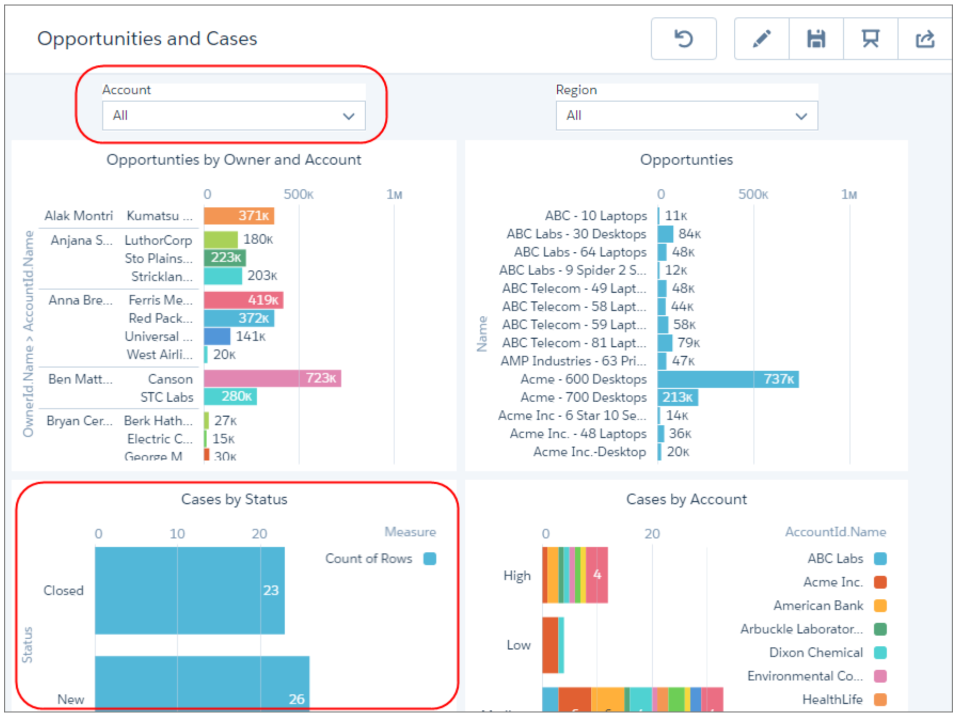
```
q = filter q by {{column(stepFoo.selection, ["grouping"]).asEquality("bar")}};
```

If the selection returns `first` and `second`, the filter becomes:

```
q = filter q by bar in ["first","second"];
```

Filter Example with the 'in' Operator (Compact Form)

Let's say you want to filter the Case by Status widget in the following dashboard based on the account selected in the Account list widget.



Faceting doesn’t work in this case because the queries on these widgets are based on different datasets. To enable filtering, create an interaction in the Cases by Status widget’s query (`Status_1`) based on the selection in the Account widget’s query (`AccountId_Name_1`). This interaction compares the value of the `AccountId.Name` field in the `Status_1` query to the selected values in the `AccountId.Name` field of the `AccountId_Name_1` query. Because there can be multiple selected account names, we’ll use the 'in' operator.

```
"steps": {
 "Status_1": {
  "datasets": [{
   "id": "0FbB00000000rlDKAQ",
   "label": "CasesAccounts",
   "name": "CasesAccounts",
   "url": "/services/data/v38.0/wave/datasets/0FbB00000000rlDKAQ"
  }],
  "isFacet": true,
  "isGlobal": false,
  "query": {
   "measures": [
    [
     "count",
     "*"
    ]
   ],
   "groups": [
    "Status"
   ],
   "filters": [
    [
     "AccountId.Name",
     "{{column(AccountId_Name_1.selection, [\"AccountId.Name\"]).asObject()}}",
     "in"
    ]
   ]
  },
  "type": "aggregateflex",
  "useGlobal": true,
  "visualizationParameters": {
   "visualizationType": "hbar",
   "options": {}
  }
 },
 "AccountId_Name_1": {
  "datasets": [{
   "id": "0FbB00000000rlIKAQ",
   "label": "OpptiesAccountsSICsUsers",
   "name": "OpptiesAccountsSICsUsers",
   "url": "/services/data/v38.0/wave/datasets/0FbB00000000rlIKAQ"
  }],
  "isFacet": true,
  "isGlobal": false,
  "query": {
   "measures": [
    [
     "count",
     "*"
    ]
   ],
   "groups": [
    "AccountId.Name"
   ]
  },
  "selectMode": "single",
  "type": "aggregateflex",
  "useGlobal": false,
  "visualizationParameters": {
   "options": {}
  }
 }
...
```

Filter Example with an Inequality Operator (SAQL Form)

Let's say you have the following results from a source query.

```
[ {grouping: "first", measure: 19} {grouping: "second", measure: 32} ]
```

You can create a filter interaction using on an inequality operator.

```
q = filter q by bar > {{cell(queryFoo.selection, 1, "measure").asString()}};
```

After evaluating the interaction, the filter becomes:

```
q = filter q by bar > 32;
```

Filter Example with the 'matches' Operator (SAQL Form)

Let's say you have the following results from a source query.

```
[ {grouping: "first", measure: 19} {grouping: "second", measure: 32} ]
```

You can create a filter interaction using on the matches operator.

```
q = filter q by bar matches "{{cell(queryFoo.selection, 1, "grouping").asString()}}";
```

After evaluating the interaction, the filter results to this.

```
q = filter q by bar matches "second";
```

Range Filters

Use the `asRange()` serialization function to bind filters based on numeric ranges.

Let's look at some examples with inclusive ranges.

The source query for an interaction produces the following results.

```
[ {grouping: "first", measure: 19} {grouping: "second", measure: 32} ]
```

You can bind the filter using the following syntax.

```
q = filter q by {{row(stepFoo.selection, [0], ["min", "max"]).asRange("bar")}};
```

After evaluating the interaction, CRM Analytics produces the following range filter.

```
q = filter q by bar >= 19 && bar <= 32;
```

Date Range Filters

Use the `asDateRange()` serialization function to bind filters based on date ranges. You can create filters using absolute or relative date ranges.

If the input data is a one-dimensional array with two elements:

- And both elements are numbers, CRM Analytics assumes the numbers are a epoch times. `[1016504910000, 1016504910000]` results in `fieldName in [dateRange([2002,3,19], [2010,8,12])]`.
- Otherwise, the first element is used as the minimum and the second element is used as the maximum. `["current day", "1 month ahead"]` results in `fieldName in ["current day".."1 month ahead"]`. If one of the elements is null, the date range is open-ended. `["1 month ago", null]` results in `fieldName in ["1 month ago"..]`.

If the input data is a two-dimensional array where the outer array has two elements:

- And both nested arrays have two elements, CRM Analytics assumes the data is in the relative date array format. `[["year", -2], ["year", 1]]` results in `fieldName in ["2 years ago".."1 year ahead"]`.
- And both nested arrays have 3 elements, the nested arrays are passed to the SAQL `dateRange()` function. `[[2015, 2, 1], [2016, 2, 1]]` results in `fieldName in [dateRange([2015,2,1], [2016,2,1])]`.

If the input data is `null`, the result is `fieldName in all`, which doesn't filter anything.

Binding to a Date Filter Widget

For instance, let's say you make a selection in a date widget that returns the following absolute date range (in epoch format).

```
[ {min: 1016504910000, max: 1281655993000} ]
```

You can create a filter using the returned selection data.

```
q = filter q by {{row(queryFoo.selection, [0], ["min", "max"]).asDateRange("date(year,
month, day)")}};
```

After evaluating the binding, CRM Analytics produces the following date range filter.

```
q = filter q by date(year, month, day) in [dateRange([2002,3,19], [2010,8,12])];
```

What about relative dates? Assume the date widget returns the following relative dates based on your selection.

```
[ {min: ["quarter", -2], max: ["quarter", 3]} ]
```

After evaluation, the following date range filter results.

```
q = filter q by date(year, month, day) in ["2 quarters ago".."3 quarters ahead"];
```

Binding to a Custom List of Date Ranges

It's common to filter based on a custom set of date ranges. To accomplish this, create a custom query with rows for each custom date range. You can specify ranges using absolute or relative dates.

To do this with absolute ranges, the results of the custom query must return absolute dates.

```
[
    {label: "8/30/15 - 8/30/16", range: [[2015, 8, 30], [2016, 8, 30]]}
    {label: "7/30/16 - 8/30/16", range: [[2016, 7, 30], [2016, 8, 30]]}
]
```

You can create the filter based on the selected value of the source query.

```
q = filter q by {{cell(queryFoo.selection, 0, "range").asDateRange("date(year, month,
day)")}};
```

After CRM Analytics evaluates the binding, the filter becomes this.

```
q = filter q by date(year, month, day) in [dateRange([2015, 8, 30], [2016, 8, 30])];
```

To do this with relative ranges, the source query results must look like this.

```
[
    {"label": "YTD", "range": ["1 year ago", "current day"]}
    {"label": "MTD", "range": ["1 month ago", "current day"]}
    {"label": "Everything up to today", "range": [null, "current day"]}
]
```

You can use the following binding to create a filter based on the selected value of the source query.

```
q = filter q by {{cell(queryFoo.selection, 0, "range").asDateRange("date(year, month,
day)")}};
```

After CRM Analytics evaluates the binding, the filter becomes:

```
q = filter q by date(year, month, day) in ["1 year ago".."current day"];
```

You can also create an open-ended range filter by specifying null as one of the relative date keywords in the source query. The bound filter looks like this.

```
q = filter q by {{cell(queryFoo.selection, 2, "range").asDateRange("date(year, month,
day)")}};
```

After CRM Analytics evaluates the binding, the filter becomes:

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

Note: The SAQL function `date_to_epoch()` returns epoch seconds, but date range filters bindings require milliseconds.

Projection Interactions

Use the `asProjection()` serialization function to specify the projection of a field in a SAQL query.

Given the following data from a source query:

```
[
    {expression: "first", alias: "foo"}
    {expression: "second", alias: "bar"}
]
```

You can bind the projection of a field in a target query.

```
q = foreach q generate {{row(stepFoo.selection, [0], ["expression",
"alias"]).asProjection()}};
```

After CRM Analytics evaluates the interaction, the projection becomes:

```
q = foreach q generate first as 'foo';
```

To return all rows in the interaction, create the following filter.

```
q = foreach q generate {{row(stepFoo.selection, [], ["expression",
"alias"]).asProjection()}};
```

After CRM Analytics evaluates the interaction, the filter becomes:

```
q = foreach q generate first as 'foo', second as 'bar';
```

Group Interactions

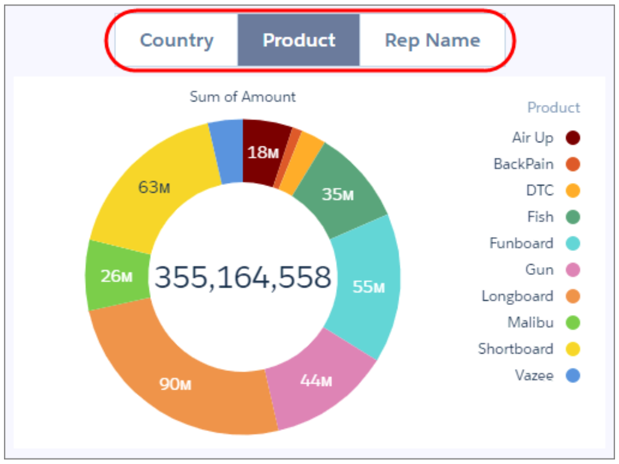
Bind the grouping to allow the dashboard viewer to select which dimensions to group the results by. For example, you can show different groupings in a chart based on the selection in a toggle widget.

To dynamically set the grouping in a query based on a selection, bind the `groups` property in the query. If the query is used for a chart, also bind the corresponding widget property under `columnMap` to identify the chart attribute affected by selected grouping. Some charts accept multiple groupings and use them differently. For example, the stacked bar chart can have two groupings, one for the vertical axis and one used to segment the bars. The `columnMap` widget-level property has subproperties that specify which grouping to use for each of these chart attributes.

To dynamically set the grouping in a query based on a selection, complete the following tasks.

- Bind the `groups` property of the query.
- If the query is used for a chart created during or after Spring '18, replace the `columnMap` section of the widget with an empty `columns` array. Why? Because when you change the query, the set of fields will likely be different from what's in the `columnMap` section. When you replace the `columnMap` property with an empty `columns` array, the system remaps the columns based on the new query definition.

Let's look at an example. Let's bind the grouping for this donut chart based on the selection in the toggle widget.



Note: Dashboard selections automatically reset each time you change your query grouping. For example, if you drill into Air Up and then switch your grouping to Country, the donut chart resets and Air Up is no longer selected.

The toggle widget uses the following custom query.

```
"GroupingsController_1": {
    "type": "staticflex",
    "values": [
        {
            "display": "Country",
            "value": "Country"
        },
        {
            "display": "Product",
            "value": "Product"
        },
        {
            "display": "Rep Name",
            "value": "Referral"
        }
    ],
    "start": {
        "display": [ "Product" ]
    },
    "broadcastFacet": true,
    "groups": [],
    "label": "GroupingsController",
    "numbers": [],
    "selectMode": "singlerequired",
    "strings": []
}
```

Each toggle option has one display label (`display`) that appears in the toggle. It also has one value (`value`) that determines the grouping.

Let's bind the `value` field of the custom query (`GroupingsController_1`) to the grouping in the donut chart's query (`PieByProduct_2`). Any selection in the custom query passes the grouping to the `PieByProduct_2` query.

```
"PieByProduct_2": {
    "label": "PieByProduct",
    "query": {
        "measures": [[
            "sum",
            "Amount"
        ]],
        "groups": [
            "{{ cell(GroupingsController_1.selection, 0, \"value\").asString() }}"
        ]
    },
    "broadcastFacet": true,
    "isGlobal": false,
    "receiveFacet": true,
    "selectMode": "single",
    "type": "aggregateflex",
    "useGlobal": true,
    "visualizationParameters": {
        "type": "chart",
        "parameters": {

            ...

        },
        "options": {}
    },
    "datasets": [
        {
            "id": "0FbB00000000q5gKAA",
            "label": "Flexy Sales",
            "name": "Flexy_Sales",
            "url": "/services/data/v42.0/wave/datasets/0FbB00000000q5gKAA"
        }
    ]
}
```

When you create the donut chart, by default, the widget (`chart_3`) contains the `columnMap` section that maps measures and groupings to chart attributes.

```
"chart_3": {
    "type": "chart",
    "parameters": {
        "visualizationType": "pie",
        "step": "PieByProduct_2",
        "theme": "wave",
        "columnMap": {
            "trellis": [],
            "dimension": [
                "{{ cell(GroupingsController_1.selection, 0, \"value\").asString() }}"
            ],
            "plots": [ "sum_Amount" ]
        },
        ...

    }
}
```

Note: The properties under the `columnMap` property vary based on the chart type.

To enable the interaction to work, replace the `columnMap` section with an empty `columns` array.

```
"chart_3": {
    "type": "chart",
    "parameters": {
        "visualizationType": "pie",
        "step": "PieByProduct_2",
        "theme": "wave",
        "columns" : [],

        ...

    }
}
```

Order Interactions

Use the `asOrder()` serialization function to specify the sort order in a SAQL query.

Let's look at an example where the selection in a toggle widget determines the sort order in a SAQL query.

Given the following data from a source query:

```
[
    {order: "first", direction: "desc"}
    {order: "second", direction: "asc"}
]
```

To order by a single field, apply the following order logic. When you don't specify the direction in the query, the default is ascending.

```
q = order q by {{cell(stepFoo.selection, 1, "order").asOrder()}};
```

After CRM Analytics evaluates the interaction, the grouping becomes:

```
q = order q by 'second';
```

To order by multiple fields, use the following grouping logic.

```
q = order q by {{column(stepFoo.selection, ["order"]).asOrder()}};
```

After CRM Analytics evaluates the interaction, the grouping becomes:

```
q = order q by ('first', 'second');
```

To specify the order and the direction, use the following grouping logic.

```
q = order q by {{row(stepFoo.selection, [], ["order", "direction"]).asOrder()}};
```

After CRM Analytics evaluates the interaction, the grouping becomes:

```
q = order q by ('first' desc, 'second' asc);
```

Limit and Offset Interactions

You can also bind the limit and offset of a SAQL query. These interactions don't require data serialization functions.

Consider a source query that provides the following data.

```
[ {limit: 100, offset: 10} ]
```

To bind the limit and offset, create the following logic.

```
q = limit q {{cell(stepFoo.selection, 0, "limit").asString()}};
q = offset q {{cell(stepFoo.selection, 0, "offset").asString()}};
```

After CRM Analytics evaluates the interaction, the limit and offset become:

```
q = limit q 100; q = offset q 10;
```

For information about limits and offsets, see the CRM Analytics SAQL Developer Guide.

Measure and Group Bindings in Compact-Form and SAQL-Form Queries

Bindings can be used both in compact-form queries and SAQL-form queries.

Let's look at an example where the selections in two custom queries (`StaticSAQLMeasureNames` and `StaticSAQLGroupingNames`) determine the measure and grouping of a SAQL-form query. Notice that the bindings for both measures and groups are defined in two places. To learn more about strings, numbers, and groups fields, see saql Step Type Properties.

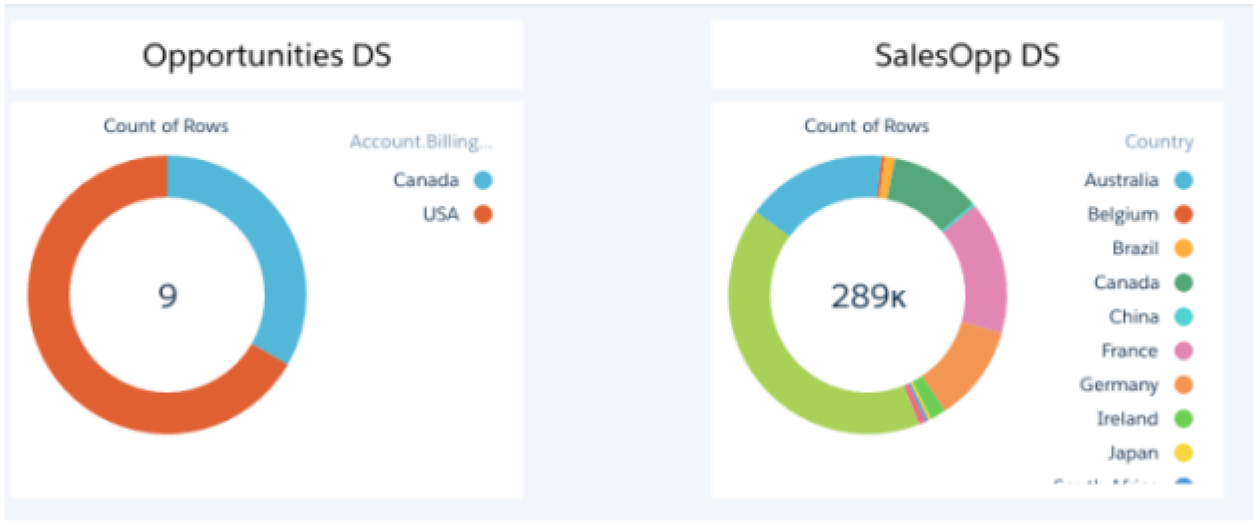
```
{
    "label": "New dashboard",
    "mobileDisabled": false,
    "state": {
        "steps": {
            "lens_1": {
                "type": "saql",
                "query": "q = load \"OpportunityWithAccount\";\nq = group q by
{{column(StaticSAQLGroupingNames.selection, [\"value\"]).asGrouping()}};\nq = foreach q
generate {{row(StaticSAQLGroupingNames.selection, [], [\"expression\",
\"alias\"]).asProjection()}}, {{row(StaticSAQLMeasureNames.selection, [], [\"expression\",
 \"alias\"]).asProjection()}};\nq = order q by 'AccountId.Industry' asc;\nq = limit q
2000;",
                "useGlobal": true,
                "numbers": "{{column(StaticSAQLMeasureNames.selection,
[\"alias\"]).asObject()}}",
                "groups": "{{column(StaticSAQLGroupingNames.selection,
[\"alias\"]).asObject()}}",
                "strings": "{{column(StaticSAQLGroupingNames.selection,
[\"alias\"]).asObject()}}",
                "visualizationParameters": {},
                "selectMode": "single",
                "broadcastFacet": true,
                "receiveFacetSource": {
                    "mode": "all",
                    “steps”: []
                }
            },
            “StaticSAQLMeasureNames”: {
                “datasets”: [],
                “dimensions”: [],
                “isFacet”: true,
                “isGlobal”: false,
                “selectMode”: “singlerequired”,
                “start”: {
                    “display”: [
                        “Total Amount”
                    ]
                },
                “type”: “staticflex”,
                “useGlobal”: true,
                “values”: [
                    {
                        “display”: “Total Amount”,
                        “cf”: [
                            “sum”,
                            “Amount”
                        ],
                        “expression”: “sum(‘Amount’)“,
                        “alias”: “sum_Amount”
                    },
                    {
                        “display”: “Avg Amount”,
                        “cf”: [
                            “avg”,
                            “Amount”
                        ],
                        “expression”: “avg(‘Amount’)“,
                        “alias”: “avg_Amount”
                    }
                ],
                “numbers”: [],
                “strings”: [],
                “groups”: [],
                “columns”: {},
                “broadcastFacet”: true
            },
            “StaticSAQLGroupingNames”: {
                “datasets”: [],
                “dimensions”: [],
                “isFacet”: true,
                “isGlobal”: false,
                “selectMode”: “multirequired”,
                “start”: {
                    “display”: [
                        “Country”
                    ]
                },
                “type”: “staticflex”,
                "useGlobal": true,
                "values": [
                    {
                        "display": "Industry",
                        "value": "AccountId.Industry",
                        "expression": "'AccountId.Industry'",
                        "alias": "AccountId.Industry"
                    },
                    {
                        "display": "Source",
                        "value": "AccountId.AccountSource",
                        "expression": "'AccountId.AccountSource'",
                        "alias": "AccountId.AccountSource"
                    }
                ],
                "numbers": [],
                "strings": [],
                "groups": [],
                "columns": {},
                "broadcastFacet": true
            }
        },
        "widgets": {
            "pillbox_3": {
                "type": "pillbox",
                "parameters": {
                    "compact": false,
                    "showActionMenu": true,
                    "exploreLink": false,
                    "fontSize": 14,
                    "textColor": "#0070D2",
                    "selectedTab": {
                        "textColor": "#FFFFFF",
                        "backgroundColor": "#0070D2",
                        "borderEdges": [
                            "all"
                        ],
                        "borderColor": "#C6D3E1",
                        "borderWidth": 1
                    },
                    "step": "StaticSAQLGroupingNames"
                }
            },
            "pillbox_4": {
                "type": "pillbox",
                "parameters": {
                    "compact": false,
                    "showActionMenu": true,
                    "exploreLink": false,
                    "fontSize": 14,
                    "textColor": "#0070D2",
                    "selectedTab": {
                        "textColor": "#FFFFFF",
                        "backgroundColor": "#0070D2",
                        “borderEdges”: [
                            “all”
                        ],
                        “borderColor”: “#C6D3E1”,
                        “borderWidth”: 1
                    },
                    “step”: “StaticSAQLMeasureNames”
                }
            },
            “chart_1”: {
                “type”: “chart”,
                “parameters”: {
                    “visualizationType”: “hbar”,
                    “title”: {
                        “label”: “”,
                        “fontSize”: 14,
                        “subtitleLabel”: “”,
                        “subtitleFontSize”: 11,
                        “align”: “center”
                    },
                    “theme”: “wave”,
                    “showValues”: true,
                    “axisMode”: “multi”,
                    “autoFitMode”: “keepLabels”,
                    “binValues”: false,
                    “bins”: {
                        “breakpoints”: {
                            “low”: 0,
                            “high”: 100
                        },
                        “bands”: {
                            “low”: {
                                “label”: “”,
                                “color”: “#B22222”
                            },
                            “medium”: {
                                “label”: “”,
                                “color”: “#FFA500"
                            },
                            “high”: {
                                “label”: “”,
                                “color”: “#008000”
                            }
                        }
                    },
                    “dimensionAxis”: {
                        “showAxis”: true,
                        “showTitle”: true,
                        “title”: “”,
                        “customSize”: “auto”,
                        “icons”: {
                            “useIcons”: false,
                            “iconProps”: {
                                “column”: “”,
                                “fit”: “cover”,
                                “type”: “round”
                            }
                        }
                    },
                    “measureAxis1": {
                        “sqrtScale”: false,
                        “showAxis”: true,
                        “customDomain”: {
                            “showDomain”: false
                        },
                        “showTitle”: true,
                        “title”: “”
                    },
                    “measureAxis2": {
                        “sqrtScale”: false,
                        “showAxis”: true,
                        “customDomain”: {
                            “showDomain”: false
                        },
                        “showTitle”: true,
                        “title”: “”
                    },
                    “legend”: {
                        “show”: true,
                        “showHeader”: true,
                        “inside”: false,
                        “descOrder”: false,
                        “position”: “right-top”,
                        “customSize”: “auto”
                    },
                    “tooltip”: {
                        “customizeTooltip”: false,
                        “showDimensions”: true,
                        “dimensions”: “”,
                        “showMeasures”: true,
                        “measures”: “”,
                        “showPercentage”: true,
                        “showNullValues”: true,
                        “showBinLabel”: true
                    },
                    “trellis”: {
                        “enable”: false,
                        “showGridLines”: true,
                        “flipLabels”: false,
                        “type”: “x”,
                        “chartsPerLine”: 4,
                        “size”: [
                            100,
                            100
                        ]
                    },
                    “applyConditionalFormatting”: true,
                    “showActionMenu”: true,
                    “exploreLink”: true,
                    “step”: “lens_1"
                }
            }
        },
        “filters”: [],
        “gridLayouts”: [
            {
                “name”: “Default”,
                “numColumns”: 12,
                “rowHeight”: “normal”,
                “version”: 1,
                “pages”: [
                    {
                        “label”: “Untitled”,
                        “name”: “36d03d4a-cdce-427e-b338-4ae29db1ba26”,
                        “widgets”: [
                            {
                                “row”: 0,
                                “column”: 6,
                                “rowspan”: 2,
                                “colspan”: 6,
                                “name”: “pillbox_3",
                                “widgetStyle”: {}
                            },
                            {
                                “row”: 2,
                                “column”: 6,
                                “rowspan”: 2,
                                “colspan”: 6,
                                “name”: “pillbox_4”,
                                “widgetStyle”: {}
                            },
                            {
                                “row”: 0,
                                “column”: 0,
                                “rowspan”: 4,
                                “colspan”: 6,
                                “name”: “chart_1",
                                “widgetStyle”: {}
                            }
                        ],
                        “navigationHidden”: false
                    }
                ],
                “selectors”: [],
                “style”: {
                    “backgroundColor”: “#F2F6FA”,
                    “gutterColor”: “#C5D3E0",
                    “cellSpacingX”: 8,
                    “cellSpacingY”: 8,
                    “fit”: “original”,
                    “alignmentX”: “left”,
                    “alignmentY”: “top”
                }
            }
        ],
        "dataSourceLinks": [],
        "widgetStyle": {
            "backgroundColor": "#FFFFFF",
            "borderEdges": [],
            "borderColor": "#E6ECF2",
            "borderWidth": 1,
            "borderRadius": 0
        }
    },
    "datasets": [
        {
            "id": "Edgemart13",
            "name": "OpportunityWithAccount",
            "label": "Opportunity With Accounts",
            "url":
"../../WaveCommon/repo/edgemarts/OpportunityWithAccount/OpportunityWithAccountEM"
        }
    ]
}
```

Note: If you bind a measure or grouping in a compact-form or SAQL-form step used for a chart created during or after Spring '18, you must also replace the `columnMap` section in the widget-level chart JSON with an empty `columns` array. For more information, see Measure Interactions and Group Interactions.

Note: If you provide an aggregate function for a measure, then the measure value must be a string, not an array.

Bind Queries from Different Datasets

You can bind queries from different datasets. For example, the following dashboard contains two charts, each based on its own dataset.



When a selection is made in the Opportunities DS chart, that selection also filters the SalesOpp DS chart because of a selection interaction. The `filters` attribute of the `Country_1` query contains a selection interaction based on the `Account_BillingCount_1` query.

Here’s the dashboard JSON for the queries and chart widgets.

```
{
    "label": "Cross-Dataset Bindings",
    "state": {
        "gridLayouts": [...],
        "layouts": [],
        "steps": {
            "Account_BillingCount_1": {
                "datasets": [
                    {
                        "id": "0Fbx000000000LzCAI",
                        "label": "Opportunities",
                        "name": "opportunity1",
                        "url": "/services/data/v38.0/wave/datasets/0Fbx000000000LzCAI"
                    }
                ],
                "isFacet": true,
                "isGlobal": false,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ],
                    "groups": [
                        "Account.BillingCountry"
                    ]
                },
                "type": "aggregateflex",
                "useGlobal": true,
                "visualizationParameters": {
                    "visualizationType": "hbar",
                    "options": {}
                }
            },
            "Country_1": {
                "datasets": [
                    {
                        "id": "0Fbx000000000NACAY",
                        "label": "SalesOpps",
                        "name": "SalesOpps",
                        "url": "/services/data/v38.0/wave/datasets/0Fbx000000000NACAY"
                    }
                ],
                "isFacet": true,
                "isGlobal": false,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ],
                    "groups": [
                        "Country"
                    ],
                    "filters": [
                        [
                            "Country",
                            "{{column(Account_BillingCount_1.selection,
[\"Account.BillingCountry\"]).asObject()}}"
                        ]
                    ]
                },
                "type": "aggregateflex",
                "useGlobal": true,
                "visualizationParameters": {
                    "visualizationType": "hbar",
                    "options": {}
                }
            }
        },
        "widgetStyle": {...},
        "widgets": {
            "text_1": {
                "parameters": {
                    "fontSize": 20,
                    "text": "Opportunities DS",
                    "textAlignment": "center",
                    "textColor": "#000000"
                },
                "type": "text"
            },
            "text_2": {
                "parameters": {
                    "fontSize": 20,
                    "text": "SalesOpp DS",
                    "textAlignment": "center",
                    "textColor": "#000000"
                },
                "type": "text"
            },
            "chart_2": {
                "parameters": {
                    "legend": {
                        "showHeader": true,
                        "show": true,
                        "position": "right-top",
                        "inside": false
                    },
                    "showMeasureTitle": true,
                    "showTotal": true,
                    "visualizationType": "pie",
                    "step": "Country_1",
                    "exploreLink": true,
                    "inner": 70,
                    "title": {
                        "label": "",
                        "subtitleLabel": "",
                        "align": "center"
                    },
                    "theme": "wave",
                    "trellis": {
                        "enable": false,
                        "type": "x",
                        "chartsPerLine": 4
                    }
                },
                "type": "chart"
            },
            "chart_1": {
                "parameters": {
                    "legend": {
                        "showHeader": true,
                        "show": true,
                        "position": "right-top",
                        "inside": false
                    },
                    "showMeasureTitle": true,
                    "showTotal": true,
                    "visualizationType": "pie",
                    "step": "Account_BillingCount_1",
                    "exploreLink": true,
                    "inner": 70,
                    "title": {
                        "label": "",
                        "subtitleLabel": "",
                        "align": "center"
                    },
                    "theme": "wave",
                    "trellis": {
                        "enable": false,
                        "type": "x",
                        "chartsPerLine": 4
                    }
                },
                "type": "chart"
            }
        }
    },
    "datasets": [
        {
            "id": "0Fbx000000000LzCAI",
            "label": "Opportunities",
            "name": "opportunity1",
            "url": "/services/data/v38.0/wave/datasets/0Fbx000000000LzCAI"
        },
        {
            "id": "0Fbx000000000NACAY",
            "label": "SalesOpps",
            "name": "SalesOpps",
            "url": "/services/data/v38.0/wave/datasets/0Fbx000000000NACAY"
        }
    ]
}
```

Bind a Custom Query with Other Queries

You can create custom queries to specify your own values for a query, instead of getting values from a query. For example, you might create a custom query to show "Top 5 Customers" and "Bottom 5 Customers" in a toggle widget. After you create the ustom query, to make it interact with the other widgets in the dashboard, manually bind the custom query to the queries of the other widgets.

For an example, see Measure Interactions.

Bind Widget Properties

In a dashboard designer dashboard only, you can implement interactions to dynamically change properties for number or chart widgets.

Highlight Values with Color Coding
You can highlight content in a widget based on selections or results in other queries. For example, color code the values of number widgets based on thresholds to draw attention to low and high numbers.

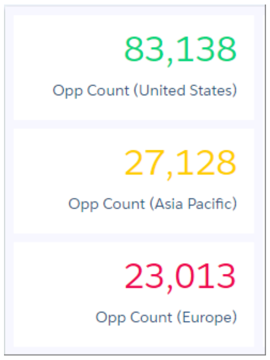
Change the Map Type Based on a Toggle Widget
You can dynamically change the map type based on selections in a toggle widget. For example, you can create a toggle that switches between two different types of maps.

Dynamically Set the Reference Line and Label
You can dynamically set a reference line and its label based on a measure from a query. For example, you might want to set the reference line to represent the sales target and then compare it against your won opportunities.

Highlight Values with Color Coding

You can highlight content in a widget based on selections or results in other queries. For example, color code the values of number widgets based on thresholds to draw attention to low and high numbers.

Let's say you want to change the colors of measures in three number widgets based on whether the numbers are high (green), medium (yellow), or low (red).



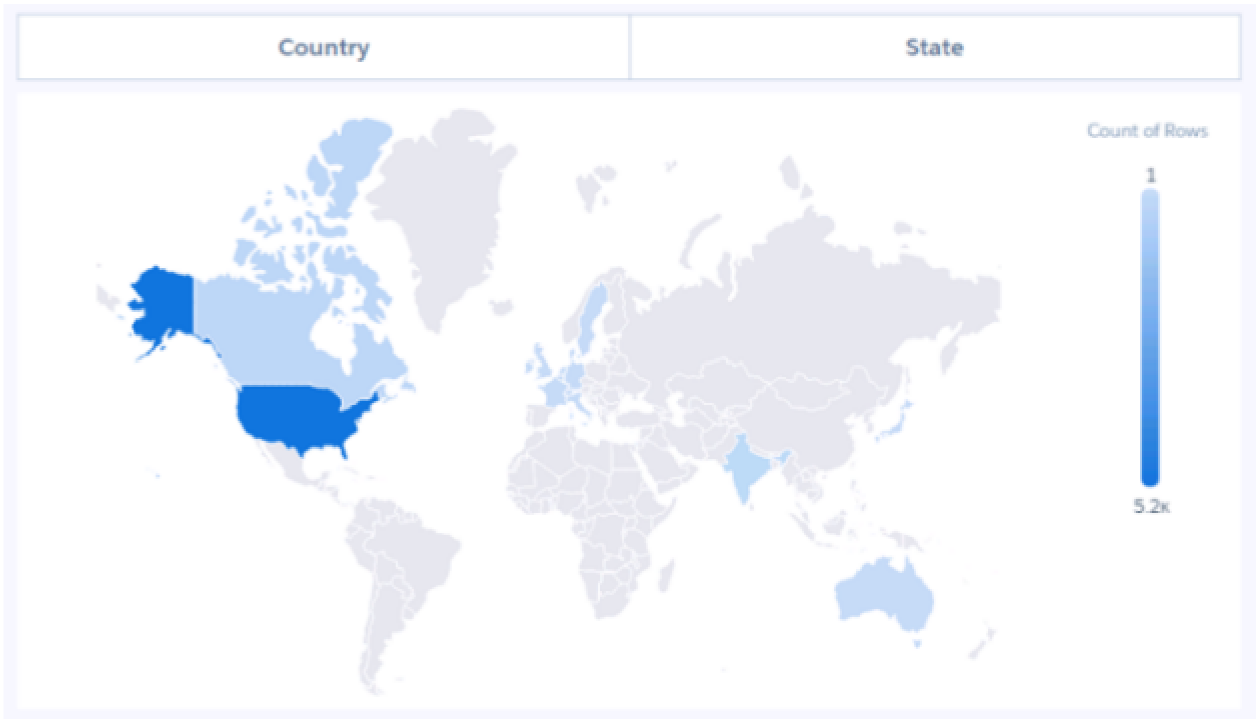
In the dashboard JSON, compute the color based on the measure of each query. Then apply the computed color to the numberColor field of each number widget.

```
{
  "label": "Sales Overview",
  "state": {
    "gridLayouts": [...],
    "layouts": [],
    "steps": {
      "color_1": {
        "type": "aggregateflex",
        "visualizationParameters": {
          "options": {}
        },
        "query": {
          "pigql": "q = load \"Opportunity_Dataset\";\n
                    q = filter q by 'Region' == \"US\";\n
                    q = group q by all;\n
                    q = foreach q generate count() as 'count',
                        (case when count() < 25000 then \"#EE0A50\"
                              when count() < 50000 then \"#F8CE00\"
                              else \"#0FD178\" end) as 'color';\n
                    q = limit q 2000;",
          "measures": [ [
            "count",
            "*",
            "count" ] ],
          "groups": [ "color" ],
          "measuresMap": {}
        },
  "isFacet": true,
  "useGlobal": true,
  "isGlobal": false,
  "datasets": [{
    "name": "Opportunity_Dataset",
     "url": "/services/data/v38.0/wave/datasets/0Fbx000000000KLCAY",
     "id": "0Fbx000000000KLCAY"
        }]
      },
      "color_2": {
        "type": "aggregateflex",
        "visualizationParameters": {
          "options": {}
        },
        "query": {
          "pigql": "q = load \"Opportunity_Dataset\";\n
                    q = filter q by 'Region' == \"AP\";\n
                    q = group q by all;\n
                    q = foreach q generate count() as 'count',
                        (case when count() < 25000 then \"#EE0A50\"
                              when count() < 50000 then \"#F8CE00\"
                              else \"#0FD178\" end) as 'color';\n
                    q = limit q 2000;",
          "measures": [ [
      "count",
      "*",
      "count"
          ] ],
          "groups": [ "color" ],
          "measuresMap": {}
        },
        "isFacet": true,
        "useGlobal": true,
        "isGlobal": false,
        "datasets": [{
          "name": "Opportunity_Dataset",
          "url": "/services/data/v38.0/wave/datasets/0Fbx000000000KLCAY",
          "id": "0Fbx000000000KLCAY"
        }]
      },
      "color_3": {
        "type": "aggregateflex",
        "visualizationParameters": {
          "options": {}
        },
        "query": {
          "pigql": "q = load \"Opportunity_Dataset\";\n
                    q = filter q by 'Region' == \"EU\";\n
                    q = group q by all;\n
                    q = foreach q generate count() as 'count',
                        (case when count() < 25000 then \"#EE0A50\"
                              when count() < 50000 then \"#F8CE00\"
                              else \"#0FD178\" end) as 'color';\n
                    q = limit q 2000;",
          "measures": [ [
            "count",
            "*",
            "count"
          ] ],
          "groups": [ "color" ],
          "measuresMap": {}
        },
        "isFacet": true,
        "useGlobal": true,
        "isGlobal": false,
        "datasets": [{
          "name": "Opportunity_Dataset",
          "url": "/services/data/v38.0/wave/datasets/0Fbx000000000KLCAY",
          "id": "0Fbx000000000KLCAY"
        }]
      }
    },
    "widgetStyle": {...},
    "widgets": {
      "number_5": {
        "type": "number",
        "parameters": {
          "step": "color_1",
          "measureField": "count",
          "textAlignment": "right",
          "compact": false,
          "exploreLink": true,
          "titleColor": "#335779",
          "titleSize": 14,
          "numberColor": "{{cell(color_1.result, 0, \"color\").asString()}}",
          "numberSize": 32,
          "title": "Opp Count (United States)"
        }
      },
      "number_6": {
        "type": "number",
        "parameters": {
          "step": "color_2",
          "measureField": "count",
          "textAlignment": "right",
          "compact": false,
          "exploreLink": true,
          "titleColor": "#335779",
          "titleSize": 14,
          "numberColor": "{{cell(color_2.result, 0, \"color\").asString()}}",
          "numberSize": 32,
          "title": "Opp Count (Asia Pacific)"
        }
      },
      "number_7": {
        "type": "number",
        "parameters": {
          "step": "color_3",
          "measureField": "count",
          "textAlignment": "right",
          "compact": false,
          "exploreLink": true,
          "titleColor": "#335779",
          "titleSize": 14,
          "numberColor": "{{cell(color_3.result, 0, \"color\").asString()}}",
          "numberSize": 32,
          "title": "Opp Count (Europe)"
        }
      }
    }
  },
  "datasets": [...]
}
```

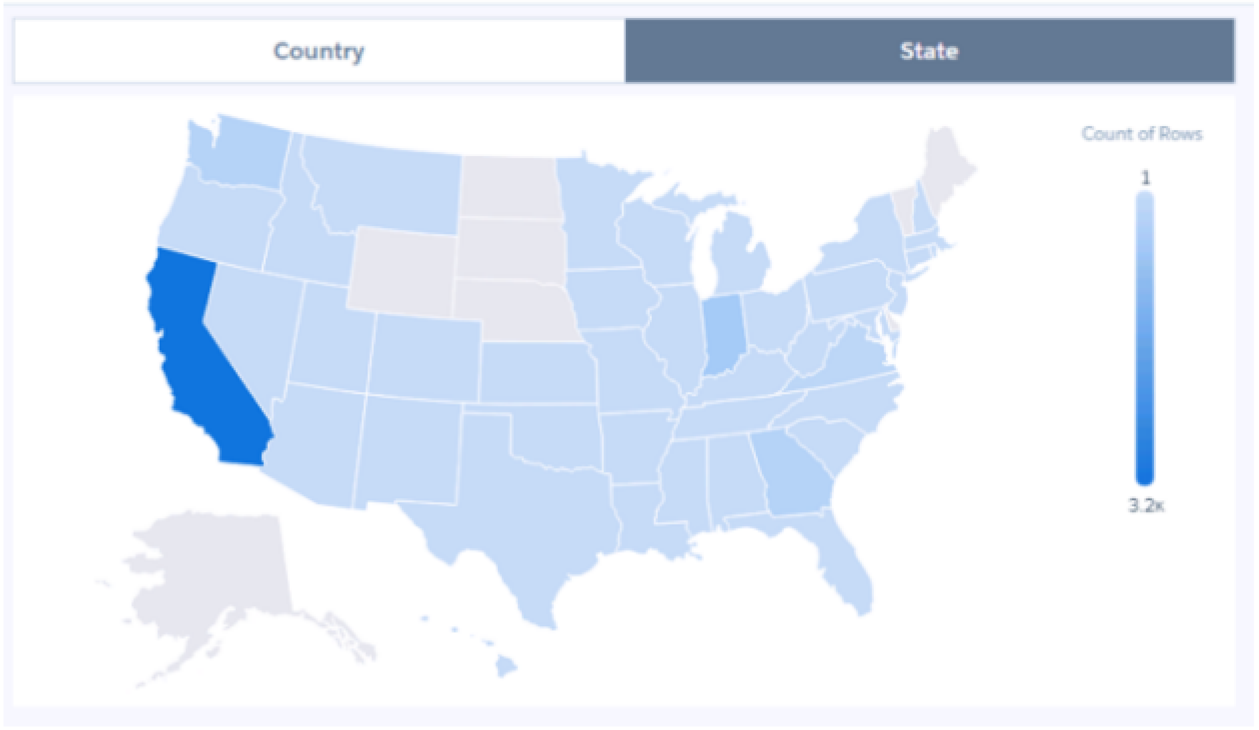
Change the Map Type Based on a Toggle Widget

You can dynamically change the map type based on selections in a toggle widget. For example, you can create a toggle that switches between two different types of maps.

Let's say you want to analyze how your company is doing both globally and specifically in the U.S. To enable this, add a toggle that allows you to switch between showing a country map of the world and a state map of the U.S. If no selection is made in the toggle, show the world map by default.



When you click the State toggle option, the dashboard shows your results for each state.



The custom query (`static_1`) provides the "Country" and "State" values that appear in the toggle widget. The chart widget has a Map Type property that allows you to select the type of map to display. To dynamically set the map type based on a selection in the custom query (`static_1`), bind the Map Type property in the query (`State__c_1`) of the chart widget to the custom query (`static_1`). For more information about maps, see Maps.

Here's the dashboard JSON.

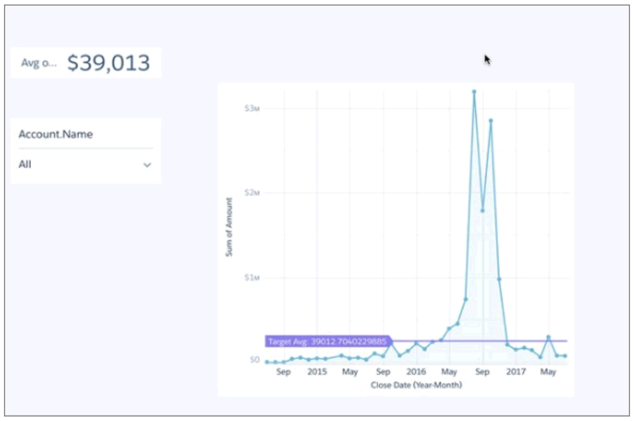
```
{
    "label": "Choropleth Binding",
    "state": {
        "gridLayouts": [...],
        "layouts": [],
        "steps": {
            "static_1": {
                "datasets": [],
                "dimensions": [],
                "isFacet": false,
                "isGlobal": false,
                "selectMode": "single",
                "type": "staticflex",
                "useGlobal": false,
                "values": [
                    {
                        "display": "Country",
                        "grouping": "Country__c",
                        "mapType": "World Countries"
                    },
                    {
                        "display": "State",
                        "grouping": "State__c",
                        "mapType": "US States"
                    }
                ],
                "visualizationParameters": {
                    "options": {}
                }
            },
            "State__c_1": {
                "datasets": [
                    {
                        "id": "0FbB000000001uGKAQ",
                        "label": "GUS Roster",
                        "name": "Roster",
                        "url": "/services/data/v38.0/wave/datasets/0FbB000000001uGKAQ"
                    }
                ],
                "isFacet": true,
                "isGlobal": false,
                "query": {
                    "measures": [
                        [
                            "count",
                            "*"
                        ]
                    ],
                    "groups": [
                        "{{coalesce(cell(static_1.selection, 0, \"grouping\"),
cell(static_1.result, 0, \"grouping\")).asString()}}"
                    ]
                },
                "type": "aggregateflex",
                "useGlobal": true,
                "visualizationParameters": {
                    "visualizationType": "hbar",
                    "options": {}
                }
            }
        },
        "widgetStyle": {...},
        "widgets": {
            "pillbox_1": {
                "parameters": {
                    "compact": false,
                    "exploreLink": false,
                    "step": "static_1"
                },
                "type": "pillbox"
            },
            "chart_1": {
                "parameters": {
                    "legend": {
                        "showHeader": true,
                        "show": true,
                        "position": "right-top",
                        "inside": false
                    },
                    "highColor": "#1674D9",
                    "lowColor": "#C5DBF7",
                    "visualizationType": "choropleth",
                    "step": "State__c_1",
                    "theme": "wave",
                    "exploreLink": true,
                    "title": {
                        "label": "",
                        "align": "center",
                        "subtitleLabel": ""
                    },
                    "trellis": {
                        "enable": false,
                        "type": "x",
                        "chartsPerLine": 4
                    },
                    "map": "{{coalesce(cell(static_1.selection, 0, \"mapType\"),
cell(static_1.result, 0, \"mapType\")).asString()}}"
                },
                "type": "chart"
            }
        }
    },
    "datasets": [...]
}
```

Dynamically Set the Reference Line and Label

You can dynamically set a reference line and its label based on a measure from a query. For example, you might want to set the reference line to represent the sales target and then compare it against your won opportunities.

Note: You can create dynamic reference lines without code. Learn more about creating reference lines.

In the following example, let's say you have a timeline chart that shows the total opportunity amount over time. The dashboard also contains a list selector that allows you to show the total amount for a particular account. To compare the total for each account against the average for all accounts, you'd like to set a reference line based on the average opportunity amount for all accounts.



To create the reference line label and its value based on the average for all accounts, add interactions in widget properties for the timeline chart (`chart_3`) as follows.

```
"referenceLines": [
     {
          "color": "#9271E8",
          "value": "{{cell(all_1.result, 0, \"avg_Amount\").asString()}}",
          "label": "Avg: {{cell(all_1.result, 0, \"avg_Amount\").asString()}}"
     }
]
```

In this example, the interaction gets the average from the query (`all_1`), which calculates the average in the number widget based on the selected account name in the list widget.

For a live example of this interaction example, install the CRM Analytics Learning Adventure app in your org. This app contains lots of interaction examples.

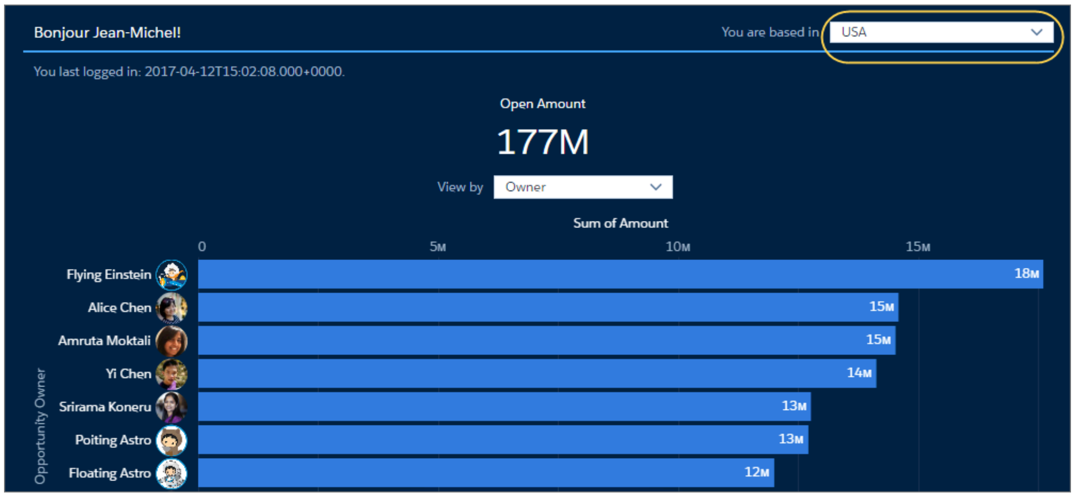
Bind the Initial Filter Selection

You can use a results interaction to dynamically set the initial selection of a query based on a characteristic of the logged-in user. For example, you can filter a region-based dashboard based on the country of the logged-in user.

Example:

Note: Only `staticflex`-, `saql`-, or `soql`-type queries can use interactions for initial filter selections.

To focus the dashboard on the logged-in user's country, configure the list widget to select the country when the dashboard first opens.



To implement this behavior, first, you need to define a query that retrieves the logged-in user's country from the Salesforce User object.

```
"UserData": {
    "groups": [],
    "numbers": [],
    "query": "SELECT Username, Country, FirstName, LastLoginDate
              FROM User WHERE Name = '!{user.name}'",
    "selectMode": "single",
    "strings": [
        "Username",
        "Country",
        "FirstName",
        "LastLoginDate"
    ],
    "type": "soql"
}
```

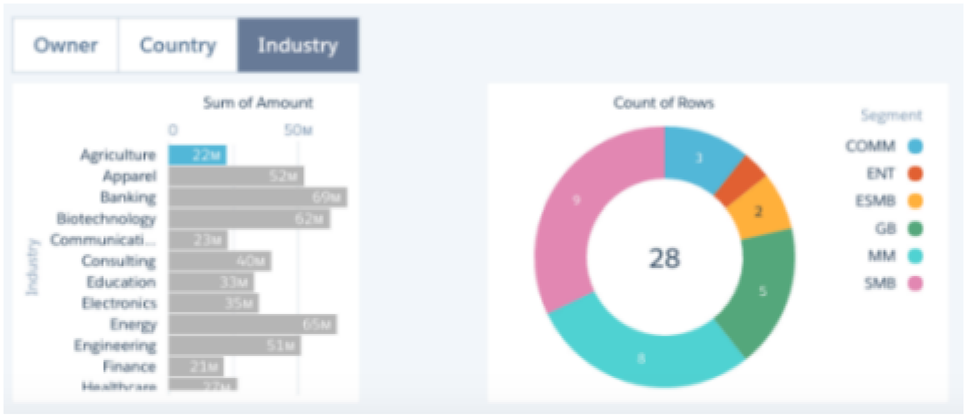
Next, bind the initial selection in the list widget's query to the logged-in user's country.

```
"Billing_Country_2": {
    "groups": [],
    "isFacet": true,
    "label": "Billing Country",
    "numbers": [],
    "query": "q = load \"Opp_Icons\";\n
              q = group q by 'Billing_Country';\n
              q = foreach q generate 'Billing_Country' as 'Billing_Country',
sum('Amount') as 'sum_Amount';\n
              q = order q by 'sum_Amount' desc;",
    "selectMode": "multi",
    "start": "{{cell(UserData.result, 0, \"Country\").asObject() }}",
    "strings": [],
    "type": "saql",
    "useGlobal": true
}
```

Create Deeper Dependencies with Nested Interactions

Nested interactions enable you to create deeper dependencies among widgets.

Example: This pie chart shows the number of opportunities for each segment. It filters the results based on the selected grouping in the toggle widget and selected value from that grouping in the horizontal bar chart.



To enable the pie chart to filter based on the value of the selected grouping, the query contains a nested interaction.

```
q = load \"Opportunities1\";\n
q = filter q by {{column(RevenueDynamicGrouping.selection,
                  column(StaticGroupingNames.selection,
                  [\"value\"])).asEquality(
                      cell(StaticGroupingNames.selection,
                      0,
                      \"value\"))
                }};\n
q = group q by 'Segment';\n
q = foreach q generate 'Segment' as 'Segment', count() as 'count';\n
q = order q by 'Segment' asc;\nq = limit q 200;
```

Note: If you bind a measure or grouping in a query used for a chart created during or after Spring '18, you must also replace the `columnMap` section in the widget-level chart JSON with an empty `columns` array. For more information, see Measure Interactions and Group Interactions.