Analytics Cloud Explorer SAQL Reference (PILOT)

Salesforce, Winter '15

@salesforcedocs

Last updated: January 2, 2015

CONTENTS

SAQL OVERVIEW

The SAQL language is a real-time query language that enables ad hoc analysis of data that's stored in datasets.

Note: SAQL is currently available through a pilot program. Any unreleased services or features referenced in this or other press releases or public statements are not currently available and may not be delivered on time or at all. Customers who purchase our services should make their purchase decisions based upon features that are currently available.

A SAQL script consists of a sequence of statements that are made up of keywords (such as `filter`, `group`, and `order`), identifiers, literals, or special characters. Statements can span multiple lines and must end with a semicolon. SAQL is procedural, which means that you describe what you want to get from your query. Then, the query engine will decide how to efficiently serve it. SAQL is compositional. Every statement has a result, and you can chain statements together. SAQL is influenced by the Pig Latin programming language, but their implementations differ and they are not compatible.

KEYWORDS

Keywords are case-sensitive and must be lowercase.

IDENTIFIERS

Identifiers are case-sensitive. They can be unquoted or quoted.

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

Quoted identifiers are wrapped in single quotes (') and can contain any character that a string can contain.

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

NUMBER LITERALS

A number literal represents a number in your script.

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

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

STRING LITERALS

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

Example:

```
"This is a string."
```

QUOTED STRING ESCAPE SEQUENCES

Strings can be escaped with the backslash character.

You can use the following string escape sequences:

Sequence	Meaning
`\b`	One backspace character
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\Z`	CTRL+Z (ASCII 26)
`\'`	One single-quote character
`\"`	One double-quote character
`\\`	One backslash character
`\0`	One ASCII null character

SPECIAL CHARACTERS

Certain characters has a special meaning in SAQL.

Character	Name	Description
`;`	Semicolon	Used to terminate statements.
`'`	Single quote	Used to quote identifiers.
`"`	Double quote	Used to quote strings.
`()`	Parentheses	Used for function calls, to enforce precedence, for order clauses, and to group expressions. Parentheses are mandatory when defining more than one group or order field.
`[]`	Brackets	Used to denote arrays. For example, this is an array of strings: `[ "this", "is", "a", "string", "array" ]` Also used for referencing a particular member of an object. For example, `em['miles']`, which is the same as `em.miles`.
`.`	Period	Used for referencing a particular member of an object. For example, `em.miles`, which is the same as `em['miles']`.
`::`	Two colons	Used to explicitly specify the dataset that a measure or dimension belongs to, by placing it between a dataset name and a column name. It is the same as using a period (.) between names. For example: `data = foreach data generate left::airline as airline`
`..`	Two periods	Used to separate a range of values. For example: `c = filter b by "the_date" in ["2011-01-01".."2011-01-31"];`

COMMENTS

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

Example:

```
a = load "myData"; --This is a comment
```

OPERATORS

Arithmetic Operators

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

On the client-side query engine, if any of the operands are NULL, an arithmetic operation returns a NULL. On the server-side query engine, if any of the operands are NULL, they are treated as if they were zero.

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

Comparison Operators

Use comparison operators to compare values.

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

On the server-side query engine, a comparison operation with a numeric operand that is NULL returns false. A comparison operation with a string operand that is NULL treats that operand as an empty string. A boolean value is always returned.

On the client-side query engine, if any of the operands are NULL, a comparison operation returns NULL.

Operator	Name	Description
==	Equals	True if the operands are equal. String comparisons that use the equals operator are case-sensitive.
!=	Not equals	True if the operands are not equal.
<	Less than	True if the left operand is less than the right operand.
<=	Less or equal	True if the left operand is less than or equal to the right operand.
>	Greater than	True if the left operand is greater than the right operand.
>=	Greater or equal	True if the left operand is greater than or equal to the right operand.

Operator	Name	Description
`matches`	Matches	True if the left operand contains the string on the right. Wildcards and regular expressions aren't supported. For example, the following query matches airport codes such as LAX, LAS, ALA, and BLA: `my_matches = filter a by origin matches "LA";`
`in`	In	If the left operand is a dimension, true if the left operand has one or more of the values in the array on the right. For example: `a1 = filter a by origin in ["ORD", "LAX", "LGA"];` If the left operand is a measure, true if the left operand is in the array on the right. You can use the `date()` function to filter by date ranges.
`in*`	In star	Used with multivalued dimensions. True for every row that has every value in an array on the right. It is a type of intersection query.
`not in`	Not in	True if the left operand is not equal to any of the values in an array on the right. It will include rows in which the origin key doesn't exist. For example: `a1 = filter a by origin not in ["ORD", "LAX", "LGA"];` Note: Only the client-side query engine currently supports the `not in` operator.
`not in*`	Not in star	Used with multivalued dimensions. True if the left operand doesn't contain every value in an array on the right. Note: Only the client-side query engine currently supports the `not in*` operator.

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

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

String Operators

Use the plus sign (+) to concatenate strings.

Operator	Description
+	Concatenate

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

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

Logical Operators

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

Logical operators can return true or false.

Operator	Name	Description
`&& (and)`	Logical AND	True if both operands are true.
`|| (or)`	Logical OR	True if either operand is true.
`!`	Logical NOT	True if the operand is false.

STATEMENTS

Load

Loads a dataset.

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

The `load` statement uses the following syntax (where <dataset> is the containerID/versionID):

```
result = load <dataset>;
```

If you're building a lens or dashboard, the UI will replace a known alias with the appropriate dataset.

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

Filter

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

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

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

The `filter` statement uses the following syntax:

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

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

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

Foreach

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

The `foreach` statement uses the following syntax:

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

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

Using Foreach with Ungrouped Data

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

Example: This example generates all carriers and the corresponding count as "flights": `a2 = foreach a1 generate carrier as carrier, miles as miles;`

Using Foreach with Grouped Data

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

Fields can only be directly accessed when the value is the same for all group members, such as the fields that were used as the grouping keys. Otherwise, the members of a group can only be accessed by using aggregate functions, rather than accessing them directly. The type of the column determines which aggregate functions can be used. For example, the `sum()` function only makes sense for numeric columns.

Example: This example demonstrates the foreach statement being used with grouped data: `z = foreach y generate day as Day, unique(origin) as uorg, count() as n;`

Group

Groups matched records.

Simple Grouping

The result of grouping is that one or more columns are added to the group. If data is grouped by a value that's NULL for a certain row, that whole row is removed from the result.

Syntax:

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

or

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

For example, to group rows by the same key:

```
a = group a by carrier;
```

You can group by multiple dimensions:

```
a = group a by (month,carrier);
```

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

Here's an alternative way to group by multiple dimensions, in separate steps:

```
a = group a by month;
a = group a by carrier;
```

Inner Cogrouping

Cogrouping means that the left and the right input are grouped independently and that the groups from the left and right are arranged side by side. Only groups that exist on both sides will appear in the results.

`inner` and `outer` are optional modifiers. If you don't specify a modifier, `inner` is used.

Syntax:

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

You can cogroup by using multiple group clauses. The result is joined by matching the group keys:

```
a = group a by carrier, b by carrier;
```

or

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

Note: Cogrouping differs from joining and then grouping the result of the join.

Several grouping and cogrouping operations can be done in sequence. The groups that result from the first cogrouping are refined by the second cogrouping operation.

Example:

```
x1 = group x by destination;
```

or

```
z = group x1 by (day,origin), y by (day,airport);
```

Groups are not hierarchical. In other words, there are no groups inside groups. Therefore, repeated grouping only splits the existing groups into smaller groups, and the smaller groups appear on the same level.

If you use aggregate functions when cogrouping, you need to specify which input side to use in the aggregate function. For example, if you have an "a" side and a "b" side that both contain a particular measure, you can use syntax that resembles the following sample:

```
sum(b['myMeasure'])
```

or

```
sum(b::myMeasure)
```

or

```
sum(b.myMeasure)
```

If you don't specify a side, the left side will be used.

Inner cogrouping can be applied across more than two sets of data, as shown in this example:

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

Outer Cogrouping

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

Syntax:

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

Example:

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

Outer cogrouping can be applied across more than two sets of data. For example, to do a left outer join from a to b, with a right join to c, you can use syntax that resembles the following sample:

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

Union

Combines multiple result sets into one result set.

The `union` statement uses the following syntax:

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

Order

Sorts by one or more attributes.

When you use the `order` statement, it isn't applied to the whole set. Instead, the rows are operated upon individually. You can specify one attribute to order by.

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

The `order` statement uses the following syntax:

```
result = order rows by attribute [ asc | desc ];
```

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

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

Limit

Limits the number of results that are returned.

Only use this statement on data that has been ordered with the order statement. The results that are returned by the `limit` statement aren't automatically ordered, and their order might change each time that statement is called. This is not a `top` or `sample` function.

You can use the `limit` statement with ungrouped data. You can also use the `limit` statement to limit grouped data by an aggregated value. For example, to find the top ten regions by revenue, you could group by region, aggregate the data using sum(revenue), order by sum(revenue) in descending order, and limit the number of results returned to the first ten results.

The expression cannot contain any columns from the input.

The `limit` statement uses the following syntax:

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

Example: This example limits the number of results that are returned to 10: `b = limit a 10;`

Offset

Paginates values from query results.

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

The `offset` statement uses the following syntax:

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

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

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

FUNCTIONS

Aggregate Functions

Use aggregate functions to perform computations on values.

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

This table lists the aggregate functions that are supported:

Aggregate Function	Description

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

`count()`	Returns the number of rows that match the query criteria. For example, to calculate the number of carriers: `q = foreach q generate 'carrier' as 'carrier', count() as 'count';`

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

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

`min()`	Returns the minimum value of a field.

Aggregate Function	Description
`max()`	Returns the maximum value of a field.
`sum()`	Returns the sum of a numeric field. `a = load "0Fbxx000000002qCAA/0Fcxx000000002WCAQ"; a = filter a by dest in ["ORD", "LAX", "ATL", "DFW", "PHX", "DEN", "LGA"]; a = group a by carrier; b = foreach a generate carrier as airline, sum(miles) as miles;`
`unique()`	Returns the count of unique values. For example, to find how many origins and destinations a carrier flies from: `a1 = group a by carrier; a2 = foreach a1 generate carrier as carrier, unique(origin) as origins, unique(dest) as destinations;`

Date Function

Use the date() function to specify to the query engine that a date is represented by the three dimensions that are specified.

date()

The dimensions must be specified in the order: year, month, day, for example, `date('year', 'month', 'day')`.

Alternatively, you can use a date column name as a parameter to the date() function, for example, `date(CreatedDate)`. In this example, during digestion, the CreatedDate is broken down into CreatedDate Year, CreatedDate Month, and CreatedDate Day. For this reason, when using the date() function, you can specify one dimension as a prefix, and the engine will automatically append the suffixes: year, month, and day.

Specify a Date Range

When filtering using the `in` operator, you can specify an array that contains a date range. For example:

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

Previously, there was a dateRange() function that took two parameters. The first parameter is an array that specifies the start date in the range. The second parameter specifies the end of the range. The dates must be specified in the order: year, month, day. For example:

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

Specify a Relative Date Range

When filtering using the `in` operator, you can specify an array that uses relative date keywords to define a range. For example:

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

The relative date keywords are:

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