Kusto Query Language (KQL) overview

Article • 06/21/2023

Kusto Query Language (KQL) is a powerful tool to explore your data and discover patterns, identify anomalies and outliers, create statistical modeling, and more. The query uses schema entities that are organized in a hierarchy similar to SQLs: databases, tables, and columns.

This article provides an explanation of the query language and offers practical exercises to get you started writing queries. To access the query environment, use the Azure Data Explorer web UI.

What is a Kusto query?

A Kusto query is a read-only request to process data and return results. The request is stated in plain text, using a data-flow model that is easy to read, author, and automate. Kusto queries are made of one or more query statements.

What is a query statement?

There are three kinds of user query statements:

- A tabular expression statement
- A let statement
- A set statement

All query statements are separated by a `;` (semicolon), and only affect the query at hand.

Note

For information about application query statements, see **Application query statements**.

The most common kind of query statement is a tabular expression **statement**, which means both its input and output consist of tables or tabular datasets. Tabular statements contain zero or more **operators**, each of which starts with a tabular input and returns a tabular output. Operators are sequenced by a `|` (pipe). Data flows, or is piped, from one operator to the next. The data is filtered or manipulated at each step and then fed into the following step.

It's like a funnel, where you start out with an entire data table. Each time the data passes through another operator, it's filtered, rearranged, or summarized. Because the piping of information from one operator to another is sequential, the query operator order is important, and can affect both results and performance. At the end of the funnel, you're left with a refined output.

Let's look at an example query.

Run the query

Kusto

```
StormEvents
| where StartTime between (datetime(2007-11-01) .. datetime(2007-12-01))
| where State == "FLORIDA"
| count
```

Count
28

ⓘ Note

KQL is case-sensitive for everything – table names, table column names, operators, functions, and so on.

This query has a single tabular expression statement. The statement begins with a reference to a table called *StormEvents* and contains several operators, where and count, each separated by a pipe. The data rows for the source table are filtered by the value of the *StartTime* column and then filtered by the value of the *State* column. In the last line, the query returns a table with a single column and a single row containing the count of the remaining rows.

To try out some more Kusto queries, see Tutorial: Write Kusto queries.

Management commands

In contrast to Kusto queries, Management commands are requests to Kusto to process or modify data or metadata. For example, the following management command creates a new Kusto table with two columns, `Level` and `Text`:

Kusto

```
.create table Logs (Level:string, Text:string)
```

Management commands have their own syntax, which isn't part of the Kusto Query Language syntax, although the two share many concepts. In particular, management commands are distinguished from queries by having the first character in the text of the command be the dot (`.`) character (which can't start a query). This distinction prevents many kinds of security attacks, simply because it prevents embedding management commands inside queries.

Not all management commands modify data or metadata. The large class of commands that start with `.show`, are used to display metadata or data. For example, the `.show tables` command returns a list of all tables in the current database.

For more information on management commands, see Management commands overview.

Next steps

- Tutorial: Learn common operators
- Tutorial: Use aggregation functions
- KQL quick reference
- SQL to Kusto cheat sheet
- Query best practices
- Query data with T-SQL

Syntax conventions for reference documentation

Article • 04/03/2023

This article outlines the syntax conventions followed in the Kusto Query Language (KQL) and management commands reference documentation.

Syntax conventions

Convention	Description
`Block`	String literals to be entered exactly as shown.
Italic	Parameters to be provided a value upon use of the function or command.
[ ] (square brackets)	Denotes that the enclosed item is optional.
[`,` ...]	Indicates that the preceding parameter can be repeated multiple times, separated by commas.
| (pipe)	Indicates that you can only use one of the syntax items separated by the pipe(s).
`;`	Query statement terminator.

Examples

Scalar function

This example shows the syntax and an example usage of the hash function, followed by an explanation of how each syntax component translates into the example usage.

Syntax

`hash(`*source* [`,` *mod*]`)`

Example usage

Kusto

```
hash("World")
```

- The name of the function, `hash`, and the opening parenthesis are entered exactly as shown.
- "World" is passed as an argument for the required *source* parameter.
- No argument is passed for the *mod* parameter, which is optional as indicated by the square brackets.
- The closing parenthesis is entered exactly as shown.

Tabular operator

This example shows the syntax and an example usage of the sort operator, followed by an explanation of how each syntax component translates into the example usage.

Syntax

T `| sort by` *column* [`asc` | `desc`] [`nulls first` | `nulls last`] [`,` ...]

Example usage

```Kusto
StormEvents
| sort by State asc, StartTime desc
```

- The StormEvents table is passed as an argument for the required *T* parameter.
- `| sort by` is entered exactly as shown. In this case, the pipe character is part of the tabular expression statement syntax, as represented by the block text. To learn more, see What is a query statement.
- The State column is passed as an argument for the required *column* parameter with the optional `asc` flag.
- After a comma, another set of arguments is passed: the StartTime column with the optional `desc` flag. The [`,` ...] syntax indicates that more argument sets may be passed but aren't required.

Working with optional parameters

To provide an argument for an optional parameter that comes after another optional parameter, you must provide an argument for the prior parameter. This requirement is because arguments must follow the order specified in the syntax. If you don't have a specific value to pass for the parameter, use an empty value of the same type.

Example of sequential optional parameters

Consider the syntax for the http_request plugin:

`evaluate` `http_request` `(` *Uri* [`,` *RequestHeaders* [`,` *Options*]] `)`

RequestHeaders and *Options* are optional parameters of type dynamic. To provide an argument for the *Options* parameter, you must also provide an argument for the *RequestHeaders* parameter. The following example shows how to provide an empty value for the first optional parameter, *RequestHeaders*, in order to be able to specify a value for the second optional parameter, *Options*.

Kusto

```
evaluate http_request ( "https://contoso.com/", dynamic({}), dynamic({
EmployeeName: Nicole }) )
```

See also

- KQL overview
- KQL quick reference

KQL quick reference

Article • 06/06/2023

This article shows you a list of functions and their descriptions to help get you started using Kusto Query Language.

Operator/Function	Description	Syntax
Filter/Search/Condition	***Find relevant data by filtering or searching***	
where	Filters on a specific predicate	`T | where Predicate`
where contains/has	`Contains`: Looks for any substring match `Has`: Looks for a specific word (better performance)	`T | where col1 contains/has "[search term]"`
search	Searches all columns in the table for the value	`[TabularSource |] search [kind=CaseSensitivity] [in (TableSources)] SearchPredicate`
take	Returns the specified number of records. Use to test a query ***Note***: `take` and `limit` are synonyms.	`T | take NumberOfRows`
case	Adds a condition statement, similar to if/then/elseif in other systems.	`case(predicate_1, then_1, predicate_2, then_2, predicate_3, then_3, else)`
distinct	Produces a table with the distinct combination of the provided columns of the input table	`distinct [ColumnName], [ColumnName]`
Date/Time	***Operations that use date and time functions***	
ago	Returns the time offset relative to the time the query executes. For example, `ago(1h)` is one hour before the current clock's reading.	`ago(a_timespan)`
format_datetime	Returns data in various date formats.	`format_datetime(datetime , format)`
bin	Rounds all values in a timeframe and groups them	`bin(value,roundTo)`

Operator/Function	Description	Syntax
Create/Remove Columns	***Add or remove columns in a table***	
print	Outputs a single row with one or more scalar expressions	`print [ColumnName =] ScalarExpression [',' ...]`
project	Selects the columns to include in the order specified	`T | project ColumnName [= Expression] [, ...]` Or `T | project [ColumnName | (ColumnName[,]) =] Expression [, ...]`
project-away	Selects the columns to exclude from the output	`T | project-away ColumnNameOrPattern [, ...]`
project-keep	Selects the columns to keep in the output	`T | project-keep ColumnNameOrPattern [, ...]`
project-rename	Renames columns in the result output	`T | project-rename new_column_name = column_name`
project-reorder	Reorders columns in the result output	`T | project-reorder Col2, Col1, Col* asc`
extend	Creates a calculated column and adds it to the result set	`T | extend [ColumnName | (ColumnName[, ...]) =] Expression [, ...]`
Sort and Aggregate Dataset	***Restructure the data by sorting or grouping them in meaningful ways***	
sort operator	Sort the rows of the input table by one or more columns in ascending or descending order	`T | sort by expression1 [asc|desc], expression2 [asc|desc], …`
top	Returns the first N rows of the dataset when the dataset is sorted using `by`	`T | top numberOfRows by expression [asc|desc] [nulls first|last]`
summarize	Groups the rows according to the `by` group columns, and calculates aggregations over each group	`T | summarize [[Column =] Aggregation [, ...]] [by [Column =] GroupExpression [, ...]]`

Operator/Function	Description	Syntax
count	Counts records in the input table (for example, T) This operator is shorthand for `summarize count()`	`T | count`
join	Merges the rows of two tables to form a new table by matching values of the specified column(s) from each table. Supports a full range of join types: `flouter`, `inner`, `innerunique`, `leftanti`, `leftantisemi`, `leftouter`, `leftsemi`, `rightanti`, `rightantisemi`, `rightouter`, `rightsemi`	`LeftTable | join [JoinParameters] ( RightTable ) on Attributes`
union	Takes two or more tables and returns all their rows	`[T1] | union [T2], [T3], …`
range	Generates a table with an arithmetic series of values	`range columnName from start to stop step step`
Format Data	***Restructure the data to output in a useful way***	
lookup	Extends the columns of a fact table with values looked-up in a dimension table	`T1 | lookup [kind = (leftouter|inner)] ( T2 ) on Attributes`
mv-expand	Turns dynamic arrays into rows (multi-value expansion)	`T | mv-expand Column`
parse	Evaluates a string expression and parses its value into one or more calculated columns. Use for structuring unstructured data.	`T | parse [kind=regex [flags=regex_flags] |simple|relaxed] Expression with * (StringConstant ColumnName [: ColumnType]) *...`
make-series	Creates series of specified aggregated values along a specified axis	`T | make-series [MakeSeriesParamters] [Column =] Aggregation [default = DefaultValue] [, ...] on AxisColumn from start to end step step [by [Column =] GroupExpression [, ...]]`

Operator/Function	Description	Syntax
let	Binds a name to expressions that can refer to its bound value. Values can be lambda expressions to create query-defined functions as part of the query. Use `let` to create expressions over tables whose results look like a new table.	`let Name = ScalarExpression | TabularExpression | FunctionDefinitionExpression`
General	***Miscellaneous operations and function***	
invoke	Runs the function on the table that it receives as input.	`T | invoke function([param1, param2])`
evaluate pluginName	Evaluates query language extensions (plugins)	`[T |] evaluate [ evaluateParameters ] PluginName ( [PluginArg1 [, PluginArg2]... )`
Visualization	***Operations that display the data in a graphical format***	
render	Renders results as a graphical output	`T | render Visualization [with (PropertyName = PropertyValue [, ...] )]`

Feedback

Add a comment in KQL

Article • 05/31/2023

Indicates user-provided text. Comments can be inserted on a separate line, nested at the end or within a KQL query or command. The engine does not evaluate the comment.

Syntax

Kusto

```
// text of comment
```

Remarks

Use the two slashes (//) for single and multi-line comments. The following table lists the keyboard shortcuts that you can use to comment or uncomment text.

Hot Key	Description
`Ctrl` + `K`, `Ctrl` + `C`	Comment current line or selected lines.
`Ctrl` + `K`, `Ctrl` + `U`	Uncomment current line or selected lines.

Example

This example returns a count of events in the New York state:

Run the query

Kusto

```
// Return the count of events in the New York state from the StormEvents
table
StormEvents
| where State == "NEW YORK" // Filter the records where the State is "NEW
YORK"
| count
```

Feedback