Query statement types

Article • 04/11/2023

A query consists of one or more **query statements**, delimited by a semicolon (`;`). At least one of these query statements must be a tabular expression statement. The tabular expression statement generates one or more tabular results. Any two statements must be separated by a semicolon. When the query has more than one tabular expression statement, the query has a batch of tabular expression statements, and the tabular results generated by these statements are all returned by the query.

Two types of query statements:

- Statements that are primarily used by users (user query statements),
- Statements that have been designed to support scenarios in which mid-tier applications take user queries and send a modified version of them to Kusto (application query statements).

Some query statements are useful in both scenarios.

> ⓘ **Note**
>
> The "effect" of a query statement starts at the point where the statement appears in the query and ends at the end of the query. Once the query completes, all its resources are released and it has no impact on future queries (other than side-effects, such as having the query recorded in a log of all queries run, or having its results cached.)

User query statements

Following is a list of user query statements:

- A let statement defines a binding between a name and an expression. Let statements can be used to break a long query into small named parts that are easier to understand.
- A set statement sets a query option that affects how the query is processed and its results returned.
- A tabular expression statement, the most important query statement, returns the "interesting" data back as results.

Application query statements

Following is a list of application query statements:

- An alias statement defines an alias to another database (in the same cluster or on a remote cluster).
- A pattern statement, which can be used by applications that are built on top of Kusto and expose the query language to their users to inject themselves into the query name resolution process.
- A query parameters statement, which is used by applications that are built on top of Kusto to protect themselves against injection attacks (similar to how command parameters protect SQL against SQL injection attacks.)
- A restrict statement, which is used by applications that are built on top of Kusto to restrict queries to a specific subset of data in Kusto (including restricting access to specific columns and records.)

Feedback

Alias statement

Article • 03/19/2023

Alias statements allow you to define an alias for databases, which can be used later in the same query.

This is useful when you're working with several clusters but want to appear as if you're working on fewer clusters. The alias must be defined according to the following syntax, where *clustername* and *databasename* are existing and valid entities.

Syntax

`alias` database *DatabaseAliasName* `=`
cluster("https://*clustername*.kusto.windows.net").database("*DatabaseName*")

Parameters

Name	Type	Required	Description
DatabaseAliasName	string	✓	An existing name or new database alias name. You can escape the name with brackets. For example, ["Name with spaces"].
DatabaseName	string	✓	The name of the database to give an alias.

> **Note**
>
> The mapped cluster-uri and the mapped database-name must appear inside double-quotes(") or single-quotes(').

Examples

In the help cluster, there's a `Samples` database with a `StormEvents` table.

First, count the number of records in that table.

Run the query

```
Kusto
```

```
StormEvents
| count
```

Output

Count
59066

Then, give an alias to the `Samples` database and use that name to check the record count of the `StormEvents` table.

Run the query

Kusto

```
alias database samplesAlias =
cluster("https://help.kusto.windows.net").database("Samples");
database("samplesAlias").StormEvents | count
```

Output

Count
59066

Create an alias name that contains spaces using the bracket syntax.

Run the query

Kusto

```
alias database ["Samples Database Alias"] =
cluster("https://help.kusto.windows.net").database("Samples");
database("Samples Database Alias").StormEvents | count
```

Output

Count
59066

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Let statement

Article • 03/26/2023

Use the `let` statement to set a variable name equal to an expression or a function, or to create views.

`let` statements are useful for:

- Breaking up a complex expression into multiple parts, each represented by a variable.
- Defining constants outside of the query body for readability.
- Defining a variable once and using it multiple times within a query.

If the variable previously represented another value, for example in nested statements, the innermost `let` statement applies.

To optimize multiple uses of the `let` statement within a single query, see Optimize queries that use named expressions.

Syntax: Scalar or tabular expressions

`let` *Name* `=` *Expression*

Parameters

Name	Type	Required	Description
Name	string	✓	The variable name. You can escape the name with brackets. For example, `["Name with spaces"]`.
Expression	string	✓	An expression with a scalar or tabular result. For example, an expression with a scalar result would be `let one=1;`, and an expression with a tabular result would be `let RecentLog = Logs | where Timestamp > ago(1h)`.

Syntax: View or function

`let` *Name* `=` [`view`] `(` [*Parameters*] `)` `{` *FunctionBody* `}`

Parameters

Name	Type	Required	Description
FunctionBody	string	✓	An expression that yields a user defined function.
`view`	string		Only relevant for a parameter-less `let` statement. When used, the `let` statement is included in queries with a `union` operator with wildcard selection of the tables/views. For an example, see Create a view or virtual table.
Parameters	string		Zero or more comma-separated tabular or scalar function parameters. For each parameter of tabular type, the parameter should be in the format *TableName*`:`*TableSchema*, in which *TableSchema* is either a comma-separated list of columns in the format *ColumnName*`:`*ColumnType* or a wildcard (`*`). If columns are specified, then the input tabular argument must contain these columns. If a wildcard is specified, then the input tabular argument can have any schema. To reference columns in the function body, they must be specified. For examples, see Tabular argument with schema and Tabular argument with wildcard. For each parameter of scalar type, provide the parameter name and parameter type in the format *Name*`:`*Type*. The name can appear in the *FunctionBody* and is bound to a particular value when the user defined function is invoked. The only supported types are `bool`, `string`, `long`, `datetime`, `timespan`, `real`, `dynamic`, and the aliases to these types.

> ⓘ Note
>
> - Tabular parameters must appear before scalar parameters.
> - Any two statements must be separated by a semicolon.

Examples

Define scalar values

The following example uses a scalar expression statement.

Kusto

```
let n = 10;  // number
let place = "Dallas";  // string
```

```
let cutoff = ago(62d); // datetime
Events
| where timestamp > cutoff
    and city == place
| take n
```

The following example binds the name `some number` using the `['name']` notation, and then uses it in a tabular expression statement.

Run the query

Kusto

```
let ['some number'] = 20;
range y from 0 to ['some number'] step 5
```

Create a user defined function with scalar calculation

This example uses the let statement with arguments for scalar calculation. The query defines function `MultiplyByN` for multiplying two numbers.

Run the query

Kusto

```
let MultiplyByN = (val:long, n:long) { val * n };
range x from 1 to 5 step 1
| extend result = MultiplyByN(x, 5)
```

Output

x	result
1	5
2	10
3	15
4	20
5	25

Create a user defined function that trims input

The following example removes leading and trailing ones from the input.

Run the query

Kusto

```
let TrimOnes = (s:string) { trim("1", s) };
range x from 10 to 15 step 1
| extend result = TrimOnes(tostring(x))
```

Output

x	result
10	0
11	
12	2
13	3
14	4
15	5

Use multiple let statements

This example defines two let statements where one statement (`foo2`) uses another (`foo1`).

Run the query

Kusto

```
let foo1 = (_start:long, _end:long, _step:long) { range x from _start to
_end step _step};
let foo2 = (_step:long) { foo1(1, 100, _step)};
foo2(2) | count
```

Output

result
50

Create a view or virtual table

This example shows you how to use a let statement to create a view or virtual table.

Run the query

Kusto

```
let Range10 = view () { range MyColumn from 1 to 10 step 1 };
let Range20 = view () { range MyColumn from 1 to 20 step 1 };
search MyColumn == 5
```

Output

$table	MyColumn
Range10	5
Range20	5

Use a materialize function

The materialize() function lets you cache subquery results during the time of query execution. When you use the `materialize()` function, the data is cached, and any subsequent invocation of the result uses cached data.

Kusto

```
let totalPagesPerDay = PageViews
| summarize by Page, Day = startofday(Timestamp)
| summarize count() by Day;
let materializedScope = PageViews
| summarize by Page, Day = startofday(Timestamp);
let cachedResult = materialize(materializedScope);
cachedResult
| project Page, Day1 = Day
| join kind = inner
(
    cachedResult
    | project Page, Day2 = Day
)
on Page
| where Day2 > Day1
| summarize count() by Day1, Day2
| join kind = inner
    totalPagesPerDay
on $left.Day1 == $right.Day
| project Day1, Day2, Percentage = count_*100.0/count_1
```

Output

Day1	Day2	Percentage
2016-05-01 00:00:00.0000000	2016-05-02 00:00:00.0000000	34.0645725975255
2016-05-01 00:00:00.0000000	2016-05-03 00:00:00.0000000	16.618368960101
2016-05-02 00:00:00.0000000	2016-05-03 00:00:00.0000000	14.6291376489636

Using nested let statements

Nested let statements are permitted, including within a user defined function expression. Let statements and arguments apply in both the current and inner scope of the function body.

Kusto

```
let start_time = ago(5h);
let end_time = start_time + 2h;
T | where Time > start_time and Time < end_time | ...
```

Tabular argument with schema

The following example specifies that the table parameter `T` must have a column `State` of type `string`. The table `T` may include other columns as well, but they can't be referenced in the function `StateState` because the aren't declared.

Run the query

Kusto

```
let StateState=(T: (State: string)) { T | extend s_s=strcat(State, State) };
StormEvents
| invoke StateState()
| project State, s_s
```

Output

State	s_s
ATLANTIC SOUTH	ATLANTIC SOUTHATLANTIC SOUTH

State	s_s
FLORIDA	FLORIDAFLORIDA
FLORIDA	FLORIDAFLORIDA
GEORGIA	GEORGIAGEORGIA
MISSISSIPPI	MISSISSIPPIMISSISSIPPI
...	...

Tabular argument with wildcard

The table parameter `T` can have any schema, and the function `CountRecordsInTable` will work.

Run the query

Kusto

```
let CountRecordsInTable=(T: (*)) { T | count };
StormEvents | invoke CountRecordsInTable()
```

Output

Count
59,066

Feedback

Pattern statement

Article • 05/23/2023

A **pattern** is a construct that maps string tuples to tabular expressions. Each pattern must *declare* a pattern name and optionally *define* a pattern mapping. Patterns that define a mapping return a tabular expression when invoked. Any two statements must be separated by a semicolon.

Empty patterns are patterns that are declared but don't define a mapping. When invoked, they return error *SEM0036* along with the details of the missing pattern definitions in the HTTP header. Middle-tier applications that provide a Kusto Query Language (KQL) experience can use the returned details as part of their process to enrich KQL query results. For more information, see Working with middle-tier applications.

Syntax

- Declare an empty pattern:

  `declare pattern` *PatternName* `;`

- Declare and define a pattern:

  `declare pattern` *PatternName* = `(`*ArgName* `:` *ArgType* [`,` ...]`)` [`[` *PathName* `:` *PathArgType* `]`]

  `{`

  `(` *ArgValue1_1* [`,` *ArgValue2_1*`,` ...] `)` [`.[` *PathValue_1* `]`] `=` `{` *expression1* `}` `;`

  [`(` *ArgValue1_2* [`,` *ArgValue2_2*`,` ...] `)` [`.[` *PathValue_2* `]`] `=` `{` *expression2* `}` `;` ...]

  `}` `;`

- Invoke a pattern:
  - *PatternName* `(` *ArgValue1* [`,` *ArgValue2* ...] `).`*PathValue*
  - *PatternName* `(` *ArgValue1* [`,` *ArgValue2* ...] `).["`*PathValue*`"]`

Parameters

Name	Type	Required	Description
PatternName	string	✓	The name of the pattern.
ArgName	string	✓	The name of the argument. Patterns can have one or more arguments.
ArgType	string	✓	The scalar data type of the *ArgName* argument. Possible values: `string`
PathName	string		The name of the path argument. Patterns can have no path or one path.
PathArgType	string		The type of the *PathArgType* argument. Possible values: `string`
ArgValue	string	✓	The *ArgName* and optional *PathName* tuple values to be mapped to an *expression*.
PathValue	string		The value to map for *PathName*.
expression	string	✓	A tabular or lambda expression that references a function returning tabular data. For example: `Logs | where Timestamp > ago(1h)`

Examples

In each of the following examples, a pattern is declared, defined, and then invoked.

Define simple patterns

The following example defines a pattern that maps states to an expression that returns its capital city.

Run the query

Kusto

```
declare pattern country = (name:string)[state:string]
{
  ("USA").["New York"] = { print Capital = "Albany" };
  ("USA").["Washington"] = { print Capital = "Olympia" };
  ("Canada").["Alberta"] = { print Capital = "Edmonton" };
};
country("Canada").Alberta
```

Output

Capital
Edmonton

Run the query

The following example defines a pattern that defines some scoped application data.

Kusto

```
declare pattern App = (applicationId:string)[scope:string]
{
    ('a1').['Data']    = { range x from 1 to 5 step 1 | project App = "App
#1", Data    = x };
    ('a1').['Metrics'] = { range x from 1 to 5 step 1 | project App = "App
#1", Metrics = rand() };
    ('a2').['Data']    = { range x from 1 to 5 step 1 | project App = "App
#2", Data    = 10 - x };
    ('a3').['Metrics'] = { range x from 1 to 5 step 1 | project App = "App
#3", Metrics = rand() };
};
union App('a2').Data, App('a1').Metrics
```

Output

App	Data	Metrics
App #2	9	
App #2	8	
App #2	7	
App #2	6	
App #2	5	
App #1		0.53674122855537532
App #1		0.78304713305654439
App #1		0.20168860732346555
App #1		0.13249123867679469
App #1		0.19388305330563443

Normalization

There are syntax variations for invoking patterns. For example, the following union returns a single pattern expression since all the invocations are of the same pattern.

Kusto

```
declare pattern app = (applicationId:string)[eventType:string]
{
    ("ApplicationX").["StopEvents"] = { database("AppX").Events | where
EventType == "StopEvent" };
    ("ApplicationX").["StartEvents"] = { database("AppX").Events | where
EventType == "StartEvent" };
};
union
  app("ApplicationX").StartEvents,
  app('ApplicationX').StartEvents,
  app("ApplicationX").['StartEvents'],
  app("ApplicationX").["StartEvents"]
```

No wildcards

There's no special treatment given to wildcards in a pattern. For example, the following query returns a single missing pattern invocation.

Kusto

```
declare pattern app = (applicationId:string)[eventType:string]
{
    ("ApplicationX").["StopEvents"] = { database("AppX").Events | where
EventType == "StopEvent" };
    ("ApplicationX").["StartEvents"] = { database("AppX").Events | where
EventType == "StartEvent" };
};
union app("ApplicationX").["*"]
| count
```

Returns semantic error

> One or more pattern references were not declared. Detected pattern references: ["app('ApplicationX').['*']"]

Work with middle-tier applications

A middle-tier application provides its users with the ability to use KQL and wants to enhance the experience by enriching the query results with augmented data from its internal service.

To this end, the application provides users with a pattern statement that returns tabular data that their users can use in their queries. The pattern's arguments are the keys the application will use to retrieve the enrichment data. When the user runs the query, the application does not parse the query itself but instead plans to leverage the error returned by an empty pattern to retrieve the keys it requires. So it prepends the query with the empty pattern declaration, sends it to the cluster for processing, and then parses the returned HTTP header to retrieve the values of missing pattern arguments. The application uses these values to look up the enrichment data and builds a new declaration that defines the appropriate enrichment data mapping. Finally, the application prepends the new definition to the user's query, resends it for processing, and returns the result it receives to the user.

Example

In the following example, a middle-tier application provides the ability to enrich queries with longitude/latitude locations. The application uses an internal service to map IP addresses to longitude/latitude locations, and provides a pattern called `map_ip_to_longlat` for this purpose. Let's suppose the application gets the following query from the user:

Kusto

```
map_ip_to_longlat("10.10.10.10")
```

The application does not parse this query and hence does not know which IP address (*10.10.10.10*) was passed to the pattern. So it prepends the user query with an empty `map_ip_to_longlat` pattern declaration and sends it for processing:

Kusto

```
declare pattern map_ip_to_longlat;
map_ip_to_longlat("10.10.10.10")
```

The application receives the following error in response.

> One or more pattern references were not declared. Detected pattern references: ["map_ip_to_longlat('10.10.10.10')"]

The application inspects the error, determines that the error indicates a missing pattern reference, and retrieves the missing IP address (*10.10.10.10*). It uses the IP address to look up the enrichment data in its internal service and builds a new pattern defining the

mapping of the IP address to the corresponding longitude and latitude data. The new pattern is prepended to the user's query and run again. This time the query succeeds because the enrichment data is now declared in the query, and the result is sent to the user.

Run the query

Kusto

```
declare pattern map_ip_to_longlat = (address:string)
{
  ("10.10.10.10") = { print Lat=37.405992, Long=-122.078515 }
};
map_ip_to_longlat("10.10.10.10")
```

Output

Lat	Long
37.405992	-122.078515

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Query parameters declaration statement

Article • 05/23/2023

Queries sent to Kusto may include a set of name or value pairs. The pairs are called *query parameters*, together with the query text itself. The query may reference one or more values, by specifying names and type, in a *query parameters declaration statement*.

Query parameters have two main uses:

- As a protection mechanism against injection attacks.
- As a way to parameterize queries.

In particular, client applications that combine user-provided input in queries that they then send to Kusto should use the mechanism to protect against the Kusto equivalent of SQL Injection attacks.

Declaring query parameters

To reference query parameters, the query text, or functions it uses, must first declare which query parameter it uses. For each parameter, the declaration provides the name and scalar type. Optionally, the parameter can also have a default value. The default is used if the request doesn't provide a concrete value for the parameter. Kusto then parses the query parameter's value, according to its normal parsing rules for that type.

Syntax

`declare query_parameters (` *Name1* `:` *Type1* [`=` *DefaultValue1*] [`,`...] `);`

Parameters

Name	Type	Required	Description
Name1	string	✓	The name of a query parameter used in the query.
Type1	string	✓	The corresponding type, such as `string` or `datetime`. The values provided by the user are encoded as strings. The appropriate parse method is applied to the query parameter to get a strongly-typed value.
DefaultValue1	string		A default value for the parameter. This value must be a literal of the appropriate scalar type.

> ⓘ Note
>
> - Like **user defined functions**, query parameters of type `dynamic` cannot have default values.
> - Let, set, and tabular statements are strung together/separated by a semicolon, otherwise they will not be considered part of the same query.

Example

Run the query

Kusto

```
declare query_parameters(maxInjured:long = 90);
StormEvents
| where InjuriesDirect + InjuriesIndirect > maxInjured
| project EpisodeId, EventType, totalInjuries = InjuriesDirect +
InjuriesIndirect
```

Output

EpisodeId	EventType	totalInjuries
12459	Winter Weather	137
10477	Excessive Heat	200
10391	Heat	187
10217	Excessive Heat	422
10217	Excessive Heat	519

Specify query parameters in a client application

The names and values of query parameters are provided as `string` values by the application making the query. No name may repeat.

The interpretation of the values is done according to the query parameters declaration statement. Every value is parsed as if it were a literal in the body of a query. The parsing is done according to the type specified by the query parameters declaration statement.

REST API

Query parameters are provided by client applications through the `properties` slot of the request body's JSON object, in a nested property bag called `Parameters`. For example, here's the body of a REST API call to Kusto that calculates the age of some user, presumably by having the application ask for the user's birthday.

JSON

```
{
    "ns": null,
    "db": "myDB",
    "csl": "declare query_parameters(birthday:datetime); print strcat(\"Your
age is: \", tostring(now() - birthday))",
    "properties": "{\"Options\":{},\"Parameters\":
{\"birthday\":\"datetime(1970-05-11)\",\"courses\":\"dynamic(['Java',
'C++'])\"}}"
}
```

Kusto .NET SDK

To provide the names and values of query parameters when using the Kusto .NET client library, one creates a new instance of the `ClientRequestProperties` object and then uses the `HasParameter`, `SetParameter`, and `ClearParameter` methods to manipulate query parameters. This class provides a number of strongly-typed overloads for `SetParameter`; internally, they generate the appropriate literal of the query language and send it as a `string` through the REST API, as described above. The query text itself must still declare the query parameters.

Kusto.Explorer

To set the query parameters sent when making a request to the service, use the **Query parameters** "wrench" icon (`ALT` + `P`).

Feedback

Restrict statement

Article • 03/12/2023

The restrict statement limits the set of table/view entities which are visible to query statements that follow it. For example, in a database that includes two tables (`A`, `B`), the application can prevent the rest of the query from accessing `B` and only "see" a limited form of table `A` by using a view.

The restrict statement's main scenario is for middle-tier applications that accept queries from users and want to apply a row-level security mechanism over those queries. The middle-tier application can prefix the user's query with a **logical model**, a set of let statements defining views that restrict the user's access to data, for example (`T | where UserId == "..."`). As the last statement being added, it restricts the user's access to the logical model only.

> ⓘ Note
>
> The restrict statement can be used to restrict access to entities in another database or cluster (wildcards are not supported in cluster names).

Syntax

`restrict access to (`*EntitySpecifiers*`)`

Parameters

Name	Type	Required	Description
EntitySpecifiers	string	✓	One or more comma-separated entity specifiers. The possible values are: - An identifier defined by a let statement as a tabular view - A table or function reference, similar to one used by a union statement - A pattern defined by a pattern declaration

> ⓘ Note
>
> - All tables, tabular views, or patterns that aren't specified by the restrict statement become "invisible" to the rest of the query.

- Let, set, and tabular statements are strung together/separated by a semicolon, otherwise they won't be considered part of the same query.

Examples

Let statement

The following example uses a let statement appearing before `restrict` statement.

Kusto

```
// Limit access to 'Test' let statement only
let Test = () { print x=1 };
restrict access to (Test);
```

Tables or functions

The following example uses references to tables or functions that are defined in the database metadata.

Kusto

```
// Assuming the database that the query uses has table Table1 and Func1
defined in the metadata,
// and other database 'DB2' has Table2 defined in the metadata

restrict access to (database().Table1, database().Func1,
database('DB2').Table2);
```

Patterns

The following example uses wildcard patterns that can match multiples of let statements or tables/functions.

Kusto

```
let Test1 = () { print x=1 };
let Test2 = () { print y=1 };
restrict access to (*);
// Now access is restricted to Test1, Test2 and no tables/functions are
accessible.

// Assuming the database that the query uses has table Table1 and Func1
```

```
defined in the metadata.
// Assuming that database 'DB2' has table Table2 and Func2 defined in the
metadata
restrict access to (database().*);
// Now access is restricted to all tables/functions of the current database
('DB2' is not accessible).

// Assuming the database that the query uses has table Table1 and Func1
defined in the metadata.
// Assuming that database 'DB2' has table Table2 and Func2 defined in the
metadata
restrict access to (database('DB2').*);
// Now access is restricted to all tables/functions of the database 'DB2'
```

Prevent user from querying other user data

The following example shows how a middle-tier application can prepend a user's query with a logical model that prevents the user from querying any other user's data.

Kusto

```
// Assume the database has a single table, UserData,
// with a column called UserID and other columns that hold
// per-user private information.
//
// The middle-tier application generates the following statements.
// Note that "username@domain.com" is something the middle-tier application
// derives per-user as it authenticates the user.
let RestrictedData = view () { Data | where UserID == "username@domain.com"
};
restrict access to (RestrictedData);
// The rest of the query is something that the user types.
// This part can only reference RestrictedData; attempting to reference Data
// will fail.
RestrictedData | summarize MonthlySalary=sum(Salary) by Year, Month
```

Kusto

```
// Restricting access to Table1 in the current database (database() called
without parameters)
restrict access to (database().Table1);
Table1 | count

// Restricting access to Table1 in the current database and Table2 in
database 'DB2'
restrict access to (database().Table1, database('DB2').Table2);
union
    (Table1),
    (database('DB2').Table2))
| count
```

```
// Restricting access to Test statement only
let Test = () { range x from 1 to 10 step 1 };
restrict access to (Test);
Test

// Assume that there is a table called Table1, Table2 in the database
let View1 = view () { Table1 | project Column1 };
let View2 = view () { Table2 | project Column1, Column2 };
restrict access to (View1, View2);

// When those statements appear before the command - the next works
let View1 = view () { Table1 | project Column1 };
let View2 = view () { Table2 | project Column1, Column2 };
restrict access to (View1, View2);
View1 |  count

// When those statements appear before the command - the next access is not
allowed
let View1 = view () { Table1 | project Column1 };
let View2 = view () { Table2 | project Column1, Column2 };
restrict access to (View1, View2);
Table1 |  count
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Set statement

Article • 05/15/2023

The `set` statement is used to set a query option for the duration of the query.

Query options control how a query executes and returns results. They can be boolean flags, which are `false` by default, or have an integer value. A query may contain zero, one, or more set statements. Set statements affect only the tabular expression statements that trail them in the program order. Any two statements must be separated by a semicolon.

- Query options can also be set programmatically in the ClientRequestProperties object.

> ⓘ **Note**
>
> Some request options that can be set in the `ClientRequestProperties` object aren't supported when set via a set statement. For more information, see **ClientRequestProperties**.

- Query options aren't formally a part of the Kusto language, and may be modified without being considered as a breaking language change.

Syntax

`set` *OptionName* [`=` *OptionValue*]

Parameters

Name	Type	Required	Description
OptionName	string	✓	The name of the query option.
OptionValue		✓	The value of the query option.

Example

```
Kusto
```

```
set querytrace;
Events | take 100
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Tabular expression statements

Article • 05/23/2023

The tabular expression statement is what people usually have in mind when they talk about queries. This statement usually appears last in the statement list, and both its input and its output consists of tables or tabular data sets. Any two statements must be separated by a semicolon.

A tabular expression statement is generally composed of *tabular data sources* such as tables, *tabular data operators* such as filters and projections, and optional *rendering operators*. The composition is represented by the pipe character (`|`), giving the statement a very regular form that visually represents the flow of tabular data from left to right. Each operator accepts a tabular data set "from the pipe", and other inputs including more tabular data sets from the body of the operator, then emits a tabular data set to the next operator that follows.

Syntax

Source `|` *Operator1* `|` *Operator2* `|` *RenderInstruction*

Parameters

Name	Type	Required	Description
Source	string	✓	A tabular data source. See Tabular data sources.
Operator	string	✓	Tabular data operators, such as filters and projections.
RenderInstruction	string		Rendering operators or instructions.

Tabular data sources

A tabular data source produces sets of records, to be further processed by tabular data operators. The following list shows supported tabular data sources:

- Table references
- The tabular range operator
- The print operator
- An invocation of a function that returns a table
- A table literal ("datatable")

Examples

Filter rows by condition

The following query counts the number of records in the `StormEvents` table that have a value of "FLORIDA" in the `State` column.

Run the query

Kusto

```
StormEvents
| where State == "FLORIDA"
| count
```

Output

Count
1042

Combine data from two tables

In the following example the join operator is used to combine records from two tabular data sources: the `StormEvents` table and the `PopulationData` table.

Run the query

Kusto

```
StormEvents
| where InjuriesDirect + InjuriesIndirect > 50
| join (PopulationData) on State
| project State, Population, TotalInjuries = InjuriesDirect +
InjuriesIndirect
```

Output

State	Population	TotalInjuries
ALABAMA	4918690	60
CALIFORNIA	39562900	61

State	Population	TotalInjuries
KANSAS	2915270	63
MISSOURI	6153230	422
OKLAHOMA	3973710	200
TENNESSEE	6886720	187
TEXAS	29363100	137

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Batches

Article • 03/19/2023

A query can include multiple tabular expression statements, as long as they're delimited by a semicolon (`;`) character. The query then returns multiple tabular results. Results are produced by the tabular expression statements and ordered according to the order of the statements in the query text.

> ⓘ Note
>
> - Prefer batching and **materialize** over using the **fork operator**.
> - Any two statements must be separated by a semicolon.

Examples

Name tabular results

The following query produces two tabular results. User agent tools can then display those results with the appropriate name associated with each (`Count of events in Florida` and `Count of events in Guam`, respectively).

Run the query

Kusto

```
StormEvents | where State == "FLORIDA" | count | as ['Count of events in Florida'];
StormEvents | where State == "GUAM" | count | as ['Count of events in Guam']
```

Share a calculation

Batching is useful for scenarios where a common calculation is shared by multiple subqueries, such as for dashboards. If the common calculation is complex, use the materialize() function and construct the query so that it will be executed only once:

Run the query

Kusto