Function types

Article • 06/05/2023

Functions are reusable queries or query parts. Kusto supports two kinds of functions:

- *Built-in functions* are hard-coded functions defined by Kusto that can't be modified by users.
- *User-defined functions*, which are divided into two types:
  - *Stored functions*: user-defined functions that are stored and managed database schema entities, similar to tables. For more information, see Stored functions. To create a stored function, use the .create function command.
  - *Query-defined functions*: user-defined functions that are defined and used within the scope of a single query. The definition of such functions is done through a let statement. For more information on how to create query-defined functions, see Create a user defined function.

  For more information on user-defined functions, see User-defined functions.

Feedback

User-defined functions

Article • 03/19/2023

User-defined functions are reusable subqueries that can be defined as part of the query itself (**query-defined functions**), or stored as part of the database metadata (**stored functions**). User-defined functions are invoked through a **name**, are provided with zero or more **input arguments** (which can be scalar or tabular), and produce a single value (which can be scalar or tabular) based on the function **body**.

A user-defined function belongs to one of two categories:

- Scalar functions
- Tabular functions, also known as views

The function's input arguments and output determine whether it's scalar or tabular, which then establishes how it might be used.

See Stored functions to create and manage entities that allow the reuse of Kusto queries or query parts.

To optimize multiple uses of the user-defined functions within a single query, see Optimize queries that use named expressions

Scalar function

- Has zero input arguments, or all its input arguments are scalar values
- Produces a single scalar value
- Can be used wherever a scalar expression is allowed
- May only use the row context in which it's defined
- Can only refer to tables (and views) that are in the accessible schema

Tabular function

- Accepts one or more tabular input arguments, and zero or more scalar input arguments, and/or:
- Produces a single tabular value

Function names

Valid user-defined function names must follow the same identifier naming rules as other entities.

The name must also be unique in its scope of definition.

> ⓘ **Note**
>
> If a stored function and a table both have the same name, then any reference to that name resolves to the stored function, not the table name. Use the **table function** to reference the table instead.

Input arguments

Valid user-defined functions follow these rules:

- A user-defined function has a strongly typed list of zero or more input arguments.
- An input argument has a name, a type, and (for scalar arguments) a default value.
- The name of an input argument is an identifier.
- The type of an input argument is either one of the scalar data types, or a tabular schema.

Syntactically, the input arguments list is a comma-separated list of argument definitions, wrapped in parenthesis. Each argument definition is specified as

Kusto

```
ArgName:ArgType [= ArgDefaultValue]
```

For tabular arguments, *ArgType* has the same syntax as the table definition (parenthesis and a list of column name/type pairs), with the addition of a solitary `(*)` indicating "any tabular schema".

For example:

Syntax	**Input arguments list description**
`()`	No arguments
`(s:string)`	Single scalar argument called `s` taking a value of type `string`
`(a:long, b:bool=true)`	Two scalar arguments, the second of which has a default value

Syntax	Input arguments list description
`(T1:(*), T2(r:real), b:bool)`	Three arguments (two tabular arguments and one scalar argument)

ⓘ Note

When using both tabular input arguments and scalar input arguments, put all tabular input arguments before the scalar input arguments.

Examples

Scalar function

Run the query

Kusto

```
let Add7 = (arg0:long = 5) { arg0 + 7 };
range x from 1 to 10 step 1
| extend x_plus_7 = Add7(x), five_plus_seven = Add7()
```

Tabular function with no arguments

Run the query

Kusto

```
let tenNumbers = () { range x from 1 to 10 step 1};
tenNumbers
| extend x_plus_7 = x + 7
```

Tabular function with arguments

Run the query

Kusto

```
let MyFilter = (T:(x:long), v:long) {
  T | where x >= v
};
MyFilter((range x from 1 to 10 step 1), 9)
```

Output

x
9
10

A tabular function that uses a tabular input with no column specified. Any table can be passed to a function, and no table columns can be referenced inside the function.

Run the query

Kusto

```
let MyDistinct = (T:(*)) {
  T | distinct *
};
MyDistinct((range x from 1 to 3 step 1))
```

Output

x
1
2
3

Declaring user-defined functions

The declaration of a user-defined function provides:

- Function **name**
- Function **schema** (parameters it accepts, if any)
- Function **body**

Note

Overloading functions isn't supported. You can't create multiple functions with the same name and different input schemas.

Tip

Lambda functions do not have a name and are bound to a name using a **let statement**. Therefore, they can be regarded as user-defined stored functions. Example: Declaration for a lambda function that accepts two arguments (a `string` called `s` and a `long` called `i`). It returns the product of the first (after converting it into a number) and the second. The lambda is bound to the name `f`:

Kusto

```
let f=(s:string, i:long) {
    tolong(s) * i
};
```

The function **body** includes:

- Exactly one expression, which provides the function's return value (scalar or tabular value).
- Any number (zero or more) of let statements, whose scope is that of the function body. If specified, the let statements must precede the expression defining the function's return value.
- Any number (zero or more) of query parameters statements, which declare query parameters used by the function. If specified, they must precede the expression defining the function's return value.

Note

Other kinds of **query statements** that are supported at the query "top level" aren't supported inside a function body. Any two statements must be separated by a semicolon.

Examples of user-defined functions

The following section shows examples of how to use user-defined functions.

User-defined function that uses a let statement

The following example shows a user-defined function (lambda) that accepts a parameter named *ID*. The function is bound to the name *Test* and makes use of three **let** statements, in which the *Test3* definition uses the *ID* parameter. When run, the output from the query is 70:

Run the query

Kusto

```
let Test = (id: int) {
  let Test2 = 10;
  let Test3 = 10 + Test2 + id;
  let Test4 = (arg: int) {
      let Test5 = 20;
      Test2 + Test3 + Test5 + arg
  };
  Test4(10)
};
range x from 1 to Test(10) step 1
| count
```

User-defined function that defines a default value for a parameter

The following example shows a function that accepts three arguments. The latter two have a default value and don't have to be present at the call site.

Run the query

Kusto

```
let f = (a:long, b:string = "b.default", c:long = 0) {
  strcat(a, "-", b, "-", c)
};
print f(12, c=7) // Returns "12-b.default-7"
```

Invoking a user-defined function

The method to invoke a user-defined function depends on the arguments that the function expects to receive. The following sections cover how to invoke a UDF without arguments, invoke a UDF with scalar arguments, and invoke a UDF with tabular arguments.

Invoke a UDF without arguments

A user-defined function that takes no arguments and can be invoked either by its name, or by its name and an empty argument list in parentheses.

Run the query

Kusto

```
// Bind the identifier a to a user-defined function (lambda) that takes
// no arguments and returns a constant of type long:
let a=(){123};
// Invoke the function in two equivalent ways:
range x from 1 to 10 step 1
| extend y = x * a, z = x * a()
```

Run the query

Kusto

```
// Bind the identifier T to a user-defined function (lambda) that takes
// no arguments and returns a random two-by-two table:
let T=(){
  range x from 1 to 2 step 1
  | project x1 = rand(), x2 = rand()
};
// Invoke the function in two equivalent ways:
// (Note that the second invocation must be itself wrapped in
// an additional set of parentheses, as the union operator
// differentiates between "plain" names and expressions)
union T, (T())
```

Invoke a UDF with scalar arguments

A user-defined function that takes one or more scalar arguments can be invoked by using the function name and a concrete argument list in parentheses:

Run the query

Kusto

```
let f=(a:string, b:string) {
  strcat(a, " (la la la)", b)
};
print f("hello", "world")
```

Invoke a UDF with tabular arguments

A user-defined function that takes one or more table arguments (with any number of scalar arguments) and can be invoked using the function name and a concrete argument list in parentheses:

Run the query

Kusto

```
let MyFilter = (T:(x:long), v:long) {
  T | where x >= v
};
MyFilter((range x from 1 to 10 step 1), 9)
```

You can also use the operator `invoke` to invoke a user-defined function that takes one or more table arguments and returns a table. This function is useful when the first concrete table argument to the function is the source of the `invoke` operator:

Run the query

Kusto

```
let append_to_column_a=(T:(a:string), what:string) {
    T | extend a=strcat(a, " ", what)
};
datatable (a:string) ["sad", "really", "sad"]
| invoke append_to_column_a(":-)")
```

Default values

Functions may provide default values to some of their parameters under the following conditions:

- Default values may be provided for scalar parameters only.
- Default values are always literals (constants). They can't be arbitrary calculations.
- Parameters with no default value always precede parameters that do have a default value.
- Callers must provide the value of all parameters with no default values arranged in the same order as the function declaration.
- Callers don't need to provide the value for parameters with default values, but may do so.
- Callers may provide arguments in an order that doesn't match the order of the parameters. If so, they must name their arguments.

The following example returns a table with two identical records. In the first invocation of `f`, the arguments are completely "scrambled", so each one is explicitly given a name:

Run the query

Kusto

```
let f = (a:long, b:string = "b.default", c:long = 0) {
  strcat(a, "-", b, "-", c)
};
union
  (print x=f(c=7, a=12)), // "12-b.default-7"
  (print x=f(12, c=7))    // "12-b.default-7"
```

Output

x
12-b.default-7
12-b.default-7

View functions

A user-defined function that takes no arguments and returns a tabular expression can be marked as a **view**. Marking a user-defined function as a view means that the function behaves like a table whenever a wildcard table name resolution is performed.

The following example shows two user-defined functions, `T_view` and `T_notview`, and shows how only the first one is resolved by the wildcard reference in the `union`:

Kusto

```
let T_view = view () { print x=1 };
let T_notview = () { print x=2 };
union T*
```

Restrictions

The following restrictions apply:

- User-defined functions can't pass into toscalar() invocation information that depends on the row-context in which the function is called.

- User-defined functions that return a tabular expression can't be invoked with an argument that varies with the row context.
- A function taking at least one tabular input can't be invoked on a remote cluster.
- A scalar function can't be invoked on a remote cluster.

The only place a user-defined function may be invoked with an argument that varies with the row context is when the user-defined function is composed of scalar functions only and doesn't use `toscalar()`.

Examples

Supported scalar function

The following query is supported because `f` is a scalar function that doesn't reference any tabular expression.

Run the query

Kusto

```
let Table1 = datatable(xdate:datetime)[datetime(1970-01-01)];
let Table2 = datatable(Column:long)[1235];
let f = (hours:long) { now() + hours*1h };
Table2 | where Column != 123 | project d = f(10)
```

The following query is supported because `f` is a scalar function that references the tabular expression `Table1` but is invoked with no reference to the current row context `f(10)`:

Run the query

Kusto

```
let Table1 = datatable(xdate:datetime)[datetime(1970-01-01)];
let Table2 = datatable(Column:long)[1235];
let f = (hours:long) { toscalar(Table1 | summarize min(xdate) - hours*1h) };
Table2 | where Column != 123 | project d = f(10)
```

Unsupported scalar function

The following query isn't supported because `f` is a scalar function that references the tabular expression `Table1`, and is invoked with a reference to the current row context `f(Column)`:

```Kusto
let Table1 = datatable(xdate:datetime)[datetime(1970-01-01)];
let Table2 = datatable(Column:long)[1235];
let f = (hours:long) { toscalar(Table1 | summarize min(xdate) - hours*1h) };
Table2 | where Column != 123 | project d = f(Column)
```

Unsupported tabular function

The following query isn't supported because `f` is a tabular function that is invoked in a context that expects a scalar value.

```Kusto
let Table1 = datatable(xdate:datetime)[datetime(1970-01-01)];
let Table2 = datatable(Column:long)[1235];
let f = (hours:long) { range x from 1 to hours step 1 | summarize
make_list(x) };
Table2 | where Column != 123 | project d = f(Column)
```

Features that are currently unsupported by user-defined functions

For completeness, here are some commonly requested features for user-defined functions that are currently not supported:

1. Function overloading: There's currently no way to overload a function (a way to create multiple functions with the same name and different input schema).

2. Default values: The default value for a scalar parameter to a function must be a scalar literal (constant).