Query operators

Article • 04/16/2023

A query is a read-only operation against a Kusto Engine cluster's ingested data. Queries always run in the context of a particular database in the cluster. They may also refer to data in another database, or even in another cluster.

As ad-hoc query of data is the top-priority scenario for Kusto, the Kusto Query Language syntax is optimized for non-expert users authoring and running queries over their data and being able to understand unambiguously what each query does (logically).

The language syntax is that of a data flow, where "data" means "tabular data" (data in one or more rows/columns rectangular shape). At a minimum, a query consists of source data references (references to Kusto tables) and one or more **query operators** applied in sequence, indicated visually by the use of a pipe character (`|`) to delimit operators.

For example:

Run the query

Kusto

```
StormEvents
| where State == 'FLORIDA' and StartTime > datetime(2000-01-01)
| count
```

Each filter prefixed by the pipe character `|` is an instance of an *operator*, with some parameters. The input to the operator is the table that is the result of the preceding pipeline. In most cases, any parameters are scalar expressions over the columns of the input. In a few cases, the parameters are the names of input columns, and in a few cases, the parameter is a second table. The result of a query is always a table, even if it only has one column and one row.

`T` is used in query to denote the preceding pipeline or source table.

Feedback

Was this page helpful? Yes No

as operator

Article • 06/18/2023

Binds a name to the operator's input tabular expression. This allows the query to reference the value of the tabular expression multiple times without breaking the query and binding a name through the let statement.

To optimize multiple uses of the `as` operator within a single query, see Named expressions.

Syntax

T `|` `as` [`hint.materialized` `=` *Materialized*] *Name*

Parameters

Name	Type	Required	Description
T	string	✓	The tabular expression to rename.
Name	string	✓	The temporary name for the tabular expression.
`hint.materialized`	bool		If *Materialized* is set to `true`, the value of the tabular expression will be as if it was wrapped by a materialize() function call. Otherwise, the value will be recalculated on every reference.

Note

- The name given by `as` will be used in the `withsource=` column of **union**, the `source_` column of **find**, and the `$table` column of **search**.
- The tabular expression named using the operator in a **join**'s outer tabular input (`$left`) can also be used in the join's tabular inner input (`$right`).

Examples

In the following two examples the union's generated TableName column will consist of 'T1' and 'T2'.

Run the query

Kusto

```
range x from 1 to 10 step 1
| as T1
| union withsource=TableName (range x from 1 to 10 step 1 | as T2)
```

Alternatively, you can write the same example as follows:

Run the query

Kusto

```
union withsource=TableName (range x from 1 to 10 step 1 | as T1), (range x
from 1 to 10 step 1 | as T2)
```

In the following example, the 'left side' of the join will be: `MyLogTable` filtered by `type == "Event"` and `Name == "Start"` and the 'right side' of the join will be: `MyLogTable` filtered by `type == "Event"` and `Name == "Stop"`

Kusto

```
MyLogTable
| where type == "Event"
| as T
| where Name == "Start"
| join (
    T
    | where Name == "Stop"
) on ActivityId
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

consume operator

Article • 03/09/2023

Consumes the tabular data stream handed to the operator.

The `consume` operator is mostly used for triggering the query side-effect without actually returning the results back to the caller.

The `consume` operator can be used for estimating the cost of a query without actually delivering the results back to the client. (The estimation isn't exact for various reasons; for example, `consume` is calculated distributively, so `T | consume` won't transmit the table's data between the nodes of the cluster.)

Syntax

`consume` [`decodeblocks` `=` *DecodeBlocks*]

Parameters

Name	Type	Required	Description
DecodeBlocks	bool		If set to `true`, or if the request property `perftrace` is set to `true`, the `consume` operator won't just enumerate the records at its input, but actually force each value in those records to be decompressed and decoded.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

count operator

Article • 12/14/2022

Returns the number of records in the input record set.

Syntax

T `|` `count`

Parameters

Name	Type	Required	Description
T	string	✓	The tabular input whose records are to be counted.

Returns

This function returns a table with a single record and column of type `long`. The value of the only cell is the number of records in *T*.

Example

Run the query

Kusto

```
StormEvents | count
```

See also

For information about the count() aggregation function, see count() (aggregation function).

Feedback

Was this page helpful? Yes No

datatable operator

Article • 03/23/2023

Returns a table whose schema and values are defined in the query itself.

> ⓘ **Note**
>
> This operator doesn't have a pipeline input.

Syntax

`datatable(` *ColumnName* `:` *ColumnType* [`,` ...] `[` *ScalarValue* [`,` *ScalarValue* ...] `])`

Parameters

Name	Type	Required	Description
ColumnName:*ColumnType*	string	✓	The name of column and type of data in that column that define the schema of the table.
ScalarValue	scalar	✓	The value to insert into the table. The number of values must be an integer multiple of the columns in the table. The *n*'th value must have a type that corresponds to column *n* % *NumColumns*.

Returns

This operator returns a data table of the given schema and data.

Example

Run the query

Kusto

```
datatable(Date:datetime, Event:string, MoreData:dynamic) [
    datetime(1910-06-11), "Born", dynamic({"key1":"value1",
"key2":"value2"}),
    datetime(1930-01-01), "Enters Ecole Navale", dynamic({"key1":"value3",
"key2":"value4"}),
    datetime(1953-01-01), "Published first book", dynamic({"key1":"value5",
"key2":"value6"}),
    datetime(1997-06-25), "Died", dynamic({"key1":"value7",
"key2":"value8"}),
]
| where strlen(Event) > 4
| extend key2 = MoreData.key2
```

Output

Date	Event	MoreData	key2
1930-01-01 00:00:00.0000000	Enters Ecole Navale	{ "key1": "value3", "key2": "value4" }	value4
1953-01-01 00:00:00.0000000	Published first book	{ "key1": "value5", "key2": "value6" }	value6

distinct operator

Article • 12/28/2022

Produces a table with the distinct combination of the provided columns of the input table.

Syntax

T `| distinct` *ColumnName*`[,`*ColumnName2*`, ...]`

Parameters

Name	Type	Required	Description
ColumnName	string	✓	The column name to search for distinct values.

ⓘ Note

The `distinct` operator supports providing an asterisk `*` as the group key to denote all columns, which is helpful for wide tables.

Example

Shows distinct combination of states and type of events that led to over 45 direct injuries.

Run the query

Kusto

```
StormEvents
| where InjuriesDirect > 45
| distinct State, EventType
```

Output

State	EventType
TEXAS	Winter Weather

State	EventType
KANSAS	Tornado
MISSOURI	Excessive Heat
OKLAHOMA	Thunderstorm Wind
OKLAHOMA	Excessive Heat
ALABAMA	Tornado
ALABAMA	Heat
TENNESSEE	Heat
CALIFORNIA	Wildfire

See also

If the group by keys are of high cardinalities, try `summarize by ...` with the shuffle strategy.

evaluate plugin operator

Article • 03/09/2023

Invokes a service-side query extension (plugin).

The `evaluate` operator is a tabular operator that allows you to invoke query language extensions known as **plugins**. Unlike other language constructs, plugins can be enabled or disabled. Plugins aren't "bound" by the relational nature of the language. In other words, they may not have a predefined, statically determined, output schema.

ⓘ **Note**

- Syntactically, `evaluate` behaves similarly to the **invoke operator**, which invokes tabular functions.
- Plugins provided through the evaluate operator aren't bound by the regular rules of query execution or argument evaluation.
- Specific plugins may have specific restrictions. For example, plugins whose output schema depends on the data. For example, **bag_unpack plugin** and **pivot plugin** can't be used when performing cross-cluster queries.

Syntax

[*T* `|`] `evaluate` [*evaluateParameters*] *PluginName* `(`[*PluginArgs*]`)`

Parameters

Name	Type	Required	Description
T	string		A tabular input to the plugin. Some plugins don't take any input and act as a tabular data source.
evaluateParameters	string		Zero or more space-separated evaluate parameters in the form of *Name* `=` *Value* that control the behavior of the evaluate operation and execution plan. Each plugin may decide differently how to handle each parameter. Refer to each plugin's documentation for specific behavior.
PluginName	string	✓	The mandatory name of the plugin being invoked.

Name	Type	Required	Description
PluginArgs	string		Zero or more comma-separated arguments to provide to the plugin.

Evaluate parameters

The following parameters are supported:

Name	Values	Description
`hint.distribution`	`single`, `per_node`, `per_shard`	Distribution hints
`hint.pass_filters`	`true`, `false`	Allow `evaluate` operator to passthrough any matching filters before the plugin. Filter is considered as 'matched' if it refers to a column existing before the `evaluate` operator. Default: `false`
`hint.pass_filters_column`	*column_name*	Allow plugin operator to passthrough filters referring to *column_name* before the plugin. Parameter can be used multiple times with different column names.

Plugins

The following plugins are supported:

- autocluster plugin
- azure-digital-twins-query-request plugin
- bag-unpack plugin
- basket plugin
- cosmosdb-sql-request plugin
- dcount-intersect plugin
- diffpatterns plugin
- diffpatterns-text plugin
- infer-storage-schema plugin
- ipv4-lookup plugin
- mysql-request-plugin
- narrow plugin
- pivot plugin

- preview plugin
- R plugin
- rolling-percentile plugin
- rows-near plugin
- schema-merge plugin
- sql-request plugin
- sequence-detect plugin

Distribution hints

Distribution hints specify how the plugin execution will be distributed across multiple cluster nodes. Each plugin may implement a different support for the distribution. The plugin's documentation specifies the distribution options supported by the plugin.

Possible values:

- `single`: A single instance of the plugin will run over the entire query data.
- `per_node`: If the query before the plugin call is distributed across nodes, then an instance of the plugin will run on each node over the data that it contains.
- `per_shard`: If the data before the plugin call is distributed across shards, then an instance of the plugin will run over each shard of the data.

extend operator

Article • 12/14/2022

Create calculated columns and append them to the result set.

Syntax

T `|` `extend` [*ColumnName* | `(`*ColumnName*[`,` ...]`)` `=`] *Expression* [`,` ...]

Parameters

Name	Type	Required	Description
T	string	✓	Tabular input to extend.
ColumnName	string		Name of the column to add or update.
Expression	string	✓	Calculation to perform over the input.

- If *ColumnName* is omitted, the output column name of *Expression* will be automatically generated.
- If *Expression* returns more than one column, a list of column names can be specified in parentheses. Then, *Expression*'s output columns will be given the specified names. If a list of the column names is not specified, all *Expression*'s output columns with generated names will be added to the output.

Returns

A copy of the input tabular result set, such that:

1. Column names noted by `extend` that already exist in the input are removed and appended as their new calculated values.
2. Column names noted by `extend` that do not exist in the input are appended as their new calculated values.

ⓘ Note

The `extend` operator adds a new column to the input result set, which does **not** have an index. In most cases, if the new column is set to be exactly the same as an existing table column that has an index, Kusto can automatically use the existing index. However, in some complex scenarios this propagation is not done. In such cases, if the goal is to rename a column, use the **project-rename operator** instead.

Example

Run the query

Kusto

```
StormEvents
| project EndTime, StartTime
| extend Duration = EndTime - StartTime
```

The following table shows only the first 10 results. To see the full output, run the query.

EndTime	StartTime	Duration
2007-01-01T00:00:00Z	2007-01-01T00:00:00Z	00:00:00
2007-01-01T00:25:00Z	2007-01-01T00:25:00Z	00:00:00
2007-01-01T02:24:00Z	2007-01-01T02:24:00Z	00:00:00
2007-01-01T03:45:00Z	2007-01-01T03:45:00Z	00:00:00
2007-01-01T04:35:00Z	2007-01-01T04:35:00Z	00:00:00
2007-01-01T04:37:00Z	2007-01-01T03:37:00Z	01:00:00
2007-01-01T05:00:00Z	2007-01-01T00:00:00Z	05:00:00
2007-01-01T05:00:00Z	2007-01-01T00:00:00Z	05:00:00
2007-01-01T06:00:00Z	2007-01-01T00:00:00Z	06:00:00
2007-01-01T06:00:00Z	2007-01-01T00:00:00Z	06:00:00

See also

- Use series_stats to return multiple columns

Feedback

externaldata operator

Article • 03/07/2023

The `externaldata` operator returns a table whose schema is defined in the query itself, and whose data is read from an external storage artifact, such as a blob in Azure Blob Storage or a file in Azure Data Lake Storage.

> ⓘ **Note**
>
> The `externaldata` operator supports a specific set of storage services, as listed under **Storage connection strings**.

> ⓘ **Note**
>
> The `externaldata` operator supports Shared Access Signature (SAS) key, Access key, and Azure AD Token authentication methods. For more information, see **Storage authentication methods**.

Syntax

`externaldata` `(`*columnName*`:`*columnType* [`,` ...] `)` `[` *storageConnectionString* [`,` ...] `]` [`with` `(` *propertyName* `=` *propertyValue* [`,` ...]`)`]

Parameters

Name	Type	Required	Description
columnName, *columnType*	string	✓	A list of column names and their types. This list defines the schema of the table.
storageConnectionString	string	✓	A storage connection string of the storage artifact to query.
propertyName, *propertyValue*	string		A list of optional properties that determines how to interpret the data retrieved from storage.

Properties

Property	Type	Description
format	string	The data format. If unspecified, an attempt is made to detect the data format from file extension. The default is `CSV`. All ingestion data formats are supported.
ignoreFirstRecord	bool	If set to `true`, the first record in every file is ignored. This property is useful when querying CSV files with headers.
ingestionMapping	string	Indicates how to map data from the source file to the actual columns in the operator result set. See data mappings.

Note

This operator doesn't accept any pipeline input.

Standard **query limits** apply to external data queries as well.

Returns

The `externaldata` operator returns a data table of the given schema whose data was parsed from the specified storage artifact, indicated by the storage connection string.

Examples

Fetch a list of user IDs stored in Azure Blob Storage

The following example shows how to find all records in a table whose `UserID` column falls into a known set of IDs, held (one per line) in an external storage file. Since the data format isn't specified, the detected data format is `TXT`.

Kusto

```
Users
| where UserID in ((externaldata (UserID:string) [

@"https://storageaccount.blob.core.windows.net/storagecontainer/users.txt"
      h@"?...SAS..." // Secret token needed to access the blob
    ]))
| ...
```

Query multiple data files

The following example queries multiple data files stored in external storage.

Kusto

```
externaldata(Timestamp:datetime, ProductId:string,
ProductDescription:string)
[

h@"https://mycompanystorage.blob.core.windows.net/archivedproducts/2019/01/0
1/part-00000-7e967c99-cf2b-4dbb-8c53-ce388389470d.csv.gz?...SAS...",

h@"https://mycompanystorage.blob.core.windows.net/archivedproducts/2019/01/0
2/part-00000-ba356fa4-f85f-430a-8b5a-afd64f128ca4.csv.gz?...SAS...",

h@"https://mycompanystorage.blob.core.windows.net/archivedproducts/2019/01/0
3/part-00000-acb644dc-2fc6-467c-ab80-d1590b23fc31.csv.gz?...SAS..."
]
with(format="csv")
| summarize count() by ProductId
```

The above example can be thought of as a quick way to query multiple data files without defining an external table.

> **Note**
>
> Data partitioning isn't recognized by the `externaldata` operator.

Query hierarchical data formats

To query hierarchical data format, such as `JSON`, `Parquet`, `Avro`, or `ORC`, `ingestionMapping` must be specified in the operator properties. In this example, there's a JSON file stored in Azure Blob Storage with the following contents:

JSON

```
{
  "timestamp": "2019-01-01 10:00:00.238521",
  "data": {
    "tenant": "e1ef54a6-c6f2-4389-836e-d289b37bcfe0",
    "method": "RefreshTableMetadata"
  }
}
{
  "timestamp": "2019-01-01 10:00:01.845423",
  "data": {
    "tenant": "9b49d0d7-b3e6-4467-bb35-fa420a25d324",
    "method": "GetFileList"
  }
}
...
```

To query this file using the `externaldata` operator, a data mapping must be specified. The mapping dictates how to map JSON fields to the operator result set columns:

Kusto

```
externaldata(Timestamp: datetime, TenantId: guid, MethodName: string)
[
   h@'https://mycompanystorage.blob.core.windows.net/events/2020/09/01/part-
0000046c049c1-86e2-4e74-8583-506bda10cca8.json?...SAS...'
]
with(format='multijson',
ingestionMapping='[{"Column":"Timestamp","Properties":
{"Path":"$.timestamp"}},{"Column":"TenantId","Properties":
{"Path":"$.data.tenant"}},{"Column":"MethodName","Properties":
{"Path":"$.data.method"}}]')
```

The `MultiJSON` format is used here because single JSON records are spanned into multiple lines.

For more info on mapping syntax, see data mappings.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

facet operator

Article • 12/28/2022

Returns a set of tables, one for each specified column. Each table specifies the list of values taken by its column. An additional table can be created by using the `with` clause.

Syntax

T `| facet by` *ColumnName* [`,` *ColumnName2*`,` ...] [`with (` *filterPipe* `)`]

Parameters

Name	Type	Required	Description
ColumnName	string	✓	The column name, or list of column names, to be summarized.
filterPipe	string		A query expression applied to the input table.

Returns

Multiple tables: one for the `with` clause, and one for each column.

Example

Run the query

Kusto

```
StormEvents
| where State startswith "A" and EventType has "Heavy"
| facet by State, EventType
    with
    (
    where StartTime between(datetime(2007-01-04) .. 7d)
    | project State, StartTime, Source, EpisodeId, EventType
    | take 5
    )
```

The following is the table generated by the `with` clause.

State	StartTime	Source	EpisodeId	EventType
ALASKA	2007-01-04 12:00:00.0000000	COOP Observer	2192	Heavy Snow
ALASKA	2007-01-04 15:00:00.0000000	Trained Spotter	2192	Heavy Snow
ALASKA	2007-01-04 15:00:00.0000000	Trained Spotter	2192	Heavy Snow
ALASKA	2007-01-04 15:00:00.0000000	Trained Spotter	2192	Heavy Snow
ALASKA	2007-01-06 18:00:00.0000000	COOP Observer	2193	Heavy Snow

The following table is the `State` facet output table.

State	count_State
ALABAMA	19
ARIZONA	33
ARKANSAS	1
AMERICAN SAMOA	1
ALASKA	58

The following table is the `EventType` facet output table.

EventType	count_EventType
Heavy Rain	34
Heavy Snow	78

find operator

Article • 03/14/2023

Finds rows that match a predicate across a set of tables.

The scope of the `find` can also be cross-database or cross-cluster.

```Kusto
find in (Table1, Table2, Table3) where Fruit=="apple"

find in (database('*').*) where Fruit == "apple"

find in (cluster('cluster_name').database('MyDB*'.*)) where Fruit == "apple"
```

Syntax

- `find` [`withsource`= *ColumnName*] [`in` `(`*Tables*`)`] `where` *Predicate* [`project-smart` | `project` *ColumnName*[`:` *ColumnType* `,` ...] [`,` `pack_all()`]]
- `find` *Predicate* [`project-smart` | `project` *ColumnName*[`:` *ColumnType* `,` ...] [`,` `pack_all()`]]

Parameters

Name	Type	Required	Description
ColumnName	string		By default, the output will include a column called *source_* whose values indicate which source table has contributed each row. If specified, *ColumnName* will be used instead of *source_*. After wildcard matching, if the query references tables from more than one database including the default database, the value of this column will have a table name qualified with the database. Similarly *cluster* and *database* qualifications will be present in the value if more than one cluster is referenced.
Predicate	bool	✓	This boolean expression is evaluated for each row in each input table. For more information, see predicate-syntax details.

Name	Type	Required	Description
Tables	string		Zero or more comma-separated table references. By default, `find` will look in all the tables in the current database. You can use: 1. The name of a table, such as `Events` 2. A query expression, such as `(Events | where id==42)` 3. A set of tables specified with a wildcard. For example, `E*` would form the union of all the tables in the database whose names begin with `E`.
`project-smart` or `project`	string		If not specified, `project-smart` will be used by default. For more information, see output-schema details.

Returns

Transformation of rows in *Table* [`,` *Table*, ...] for which *Predicate* is `true`. The rows are transformed according to the output schema.

Output schema

source_ column

The find operator output will always include a *source_* column with the source table name. The column can be renamed using the `withsource` parameter.

results columns

Source tables that don't contain any column used by the predicate evaluation, will be filtered out.

When you use `project-smart`, the columns that will appear in the output will be:

- Columns that appear explicitly in the predicate.
- Columns that are common to all the filtered tables.

The rest of the columns will be packed into a property bag and will appear in an additional `pack` column. A column that is referenced explicitly by the predicate and appears in multiple tables with multiple types, will have a different column in the result schema for each such type. Each of the column names will be constructed from the original column name and type, separated by an underscore.

When using `project` *ColumnName*[`:` *ColumnType* `,` ...] [`,` `pack_all()`]:

- The result table will include the columns specified in the list. If a source table doesn't contain a certain column, the values in the corresponding rows will be null.
- When specifying a *ColumnType* with a *ColumnName*, this column in the "result" will have the given type, and the values will be cast to that type if needed. The casting won't have an effect on the column type when evaluating the *Predicate*.
- When `pack_all()` is used, all the columns, including the projected columns, are packed into a property bag and appear in an additional column, by default 'column1'. In the property bag, the source column name serves as the property name and the column's value serves as the property value.

Predicate syntax

The *find* operator supports an alternative syntax for the `* has` term, and using just *term*, will search a term across all input columns.

For a summary of some filtering functions, see where operator.

Notes

- If the `project` clause references a column that appears in multiple tables and has multiple types, a type must follow this column reference in the project clause
- If a column appears in multiple tables and has multiple types and `project-smart` is in use, there will be a corresponding column for each type in the `find`'s result, as described in union
- When you use *project-smart*, changes in the predicate, in the source tables set, or in the tables schema, may result in a change to the output schema. If a constant result schema is needed, use *project* instead
- `find` scope can't include functions. To include a function in the find scope, define a let statement with view keyword.

Performance tips

- Use tables as opposed to tabular expressions. If tabular expression, the find operator falls back to a `union` query that can result in degraded performance.
- If a column that appears in multiple tables and has multiple types, is part of the project clause, prefer adding a *ColumnType* to the project clause over modifying the table before passing it to `find`.
- Add time-based filters to the predicate. Use a datetime column value or ingestion_time().

- Search in specific columns rather than a full text search.
- It's better not to reference columns that appear in multiple tables and have multiple types. If the predicate is valid when resolving such columns type for more than one type, the query will fall back to union. For example, see examples of cases where find will act as a union.

Examples

Term lookup across all tables in current database

The query finds all rows from all tables in the current database in which any column includes the word `Hernandez`. The resulting records are transformed according to the output schema. The output includes rows from the `Customers` table and the `SalesTable` table of the `ContosoSales` database.

Run the query

Kusto

```
find "Hernandez"
```

Term lookup across all tables matching a name pattern in the current database

The query finds all rows from all tables in the current database whose name starts with `C`, and in which any column includes the word `Hernandez`. The resulting records are transformed according to the output schema. Now, the output only contains records from the `Customers` table.

Run the query

Kusto

```
find in (C*) where * has "Hernandez"
```

Term lookup across all tables in all databases in the cluster

The query finds all rows from all tables in all databases in which any column includes the word `Kusto`. This query is a cross-database query. The resulting records are transformed according to the output schema.

Run the query

Kusto

```
find in (database('*').*) where * has "Kusto"
```

Term lookup across all tables and databases matching a name pattern in the cluster

The query finds all rows from all tables whose name starts with `K` in all databases whose name start with `B` and in which any column includes the word `Kusto`. The resulting records are transformed according to the output schema.

Run the query

Kusto

```
find in (database("S*").C*) where * has "Kusto"
```

Term lookup in several clusters

The query finds all rows from all tables whose name starts with `K` in all databases whose name start with `B` and in which any column includes the word `Kusto`. The resulting records are transformed according to the output schema.

Kusto

```
find in (cluster("cluster1").database("B*").K*,
cluster("cluster2").database("C*".*))
where * has "Kusto"
```

Examples of `find` output results

The following examples show how `find` can be used over two tables: *EventsTable1* and *EventsTable2*. Assume we have the next content of these two tables:

EventsTable1

Session_Id	Level	EventText	Version
acbd207d-51aa-4df7-bfa7-be70eb68f04e	Information	Some Text1	v1.0.0
acbd207d-51aa-4df7-bfa7-be70eb68f04e	Error	Some Text2	v1.0.0
28b8e46e-3c31-43cf-83cb-48921c3986fc	Error	Some Text3	v1.0.1
8f057b11-3281-45c3-a856-05ebb18a3c59	Information	Some Text4	v1.1.0

EventsTable2

Session_Id	Level	EventText	EventName
f7d5f95f-f580-4ea6-830b-5776c8d64fdd	Information	Some Other Text1	Event1
acbd207d-51aa-4df7-bfa7-be70eb68f04e	Information	Some Other Text2	Event2
acbd207d-51aa-4df7-bfa7-be70eb68f04e	Error	Some Other Text3	Event3
15eaeab5-8576-4b58-8fc6-478f75d8fee4	Error	Some Other Text4	Event4

Search in common columns, project common and uncommon columns, and pack the rest

Kusto

```
find in (EventsTable1, EventsTable2)
     where Session_Id == 'acbd207d-51aa-4df7-bfa7-be70eb68f04e' and Level ==
'Error'
     project EventText, Version, EventName, pack_all()
```

Output

source_	EventText	Version	EventName	pack_
EventsTable1	Some Text2	v1.0.0		{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Error"}
EventsTable2	Some Other Text3		Event3	{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Error"}

Search in common and uncommon columns

```
Kusto
```

```
find Version == 'v1.0.0' or EventName == 'Event1' project Session_Id,
EventText, Version, EventName
```

Output

source_	Session_Id	EventText	Version	EventName
EventsTable1	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Some Text1	v1.0.0	
EventsTable1	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Some Text2	v1.0.0	
EventsTable2	f7d5f95f-f580-4ea6-830b-5776c8d64fdd	Some Other Text1		Event1

Note: in practice, *EventsTable1* rows will be filtered with `Version == 'v1.0.0'` predicate and *EventsTable2* rows will be filtered with `EventName == 'Event1'` predicate.

Use abbreviated notation to search across all tables in the current database

```
Kusto
```

```
find Session_Id == 'acbd207d-51aa-4df7-bfa7-be70eb68f04e'
```

Output

source_	Session_Id	Level	EventText	pack_
EventsTable1	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Information	Some Text1	{"Version":"v1.0.0"}
EventsTable1	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Error	Some Text2	{"Version":"v1.0.0"}
EventsTable2	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Information	Some Other Text2	{"EventName":"Event2"}
EventsTable2	acbd207d-51aa-4df7-bfa7-be70eb68f04e	Error	Some Other Text3	{"EventName":"Event3"}

Return the results from each row as a property bag

Kusto

```
find Session_Id == 'acbd207d-51aa-4df7-bfa7-be70eb68f04e' project pack_all()
```

Output

source_	pack_
EventsTable1	{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Information", "EventText":"Some Text1", "Version":"v1.0.0"}
EventsTable1	{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Error", "EventText":"Some Text2", "Version":"v1.0.0"}
EventsTable2	{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Information", "EventText":"Some Other Text2", "EventName":"Event2"}
EventsTable2	{"Session_Id":"acbd207d-51aa-4df7-bfa7-be70eb68f04e", "Level":"Error", "EventText":"Some Other Text3", "EventName":"Event3"}

Examples of cases where `find` will act as `union`

Using a non-tabular expression as find operand

Kusto

```
let PartialEventsTable1 = view() { EventsTable1 | where Level == 'Error' };
find in (PartialEventsTable1, EventsTable2)
     where Session_Id == 'acbd207d-51aa-4df7-bfa7-be70eb68f04e'
```

Referencing a column that appears in multiple tables and has multiple types

Assume we've created two tables by running:

Kusto

```
.create tables
  Table1 (Level:string, Timestamp:datetime, ProcessId:string),
  Table2 (Level:string, Timestamp:datetime, ProcessId:int64)
```

- The following query will be executed as `union`.

Kusto

```
find in (Table1, Table2) where ProcessId == 1001
```

The output result schema will be *(Level:string, Timestamp, ProcessId_string, ProcessId_int)*.

- The following query will also be executed as `union`, but will produce a different result schema.

Kusto

```
find in (Table1, Table2) where ProcessId == 1001 project Level, Timestamp,
ProcessId:string
```

The output result schema will be *(Level:string, Timestamp, ProcessId_string)*

fork operator

Article • 03/19/2023

Runs multiple consumer operators in parallel.

Syntax

T `|` `fork` [*name*`=`]`(`*subquery*`)` [*name*`=`]`(`*subquery*`)` ...

Parameters

Name	Type	Required	Description
subquery	string	✓	A downstream pipeline of supported query operators.
name	string		A temporary name for the subquery result table.

ⓘ **Note**

- Avoid using `fork` with a single *subquery*.
- The name of the results tab will be the same name as provided with the `name` parameter or the as operator.

Supported query operators

- as
- count
- extend
- parse
- where
- take
- project
- project-away
- project-keep
- project-rename
- project-reorder
- summarize
- top

- top-nested
- sort
- mv-expand
- reduce

Returns

Multiple result tables, one for each of the *subquery* arguments.

Tips

- Use materialize as a replacement for join or union on fork legs. The input stream will be cached by materialize and then the cached expression can be used in join/union legs.

- Use batch with materialize of tabular expression statements instead of the `fork` operator.

Examples

Unnamed subqueries

Run the query

Kusto

```
StormEvents
| where State == "FLORIDA"
| fork
    ( where DeathsDirect + DeathsIndirect > 1)
    ( where InjuriesDirect + InjuriesIndirect > 1)
```

Named subqueries

In the following examples, the result tables will be named "StormsWithDeaths" and "StormsWithInjuries".

Run the query

Kusto

```
StormEvents
| where State == "FLORIDA"
| fork
    (where DeathsDirect + DeathsIndirect > 1 | as StormsWithDeaths)
    (where InjuriesDirect + InjuriesIndirect > 1 | as StormsWithInjuries)
```

Run the query

Kusto

```
StormEvents
| where State == "FLORIDA"
| fork
    StormsWithDeaths = (where DeathsDirect + DeathsIndirect > 1)
    StormsWithInjuries = (where InjuriesDirect + InjuriesIndirect > 1)
```

getschema operator

Article • 01/02/2023

Produce a table that represents a tabular schema of the input.

Syntax

T `|` `getschema`

Example

Run the query

Kusto

```
StormEvents
| getschema
```

Output

ColumnName	ColumnOrdinal	DataType	ColumnType
StartTime	0	System.DateTime	datetime
EndTime	1	System.DateTime	datetime
EpisodeId	2	System.Int32	int
EventId	3	System.Int32	int
State	4	System.String	string
EventType	5	System.String	string
InjuriesDirect	6	System.Int32	int
InjuriesIndirect	7	System.Int32	int
DeathsDirect	8	System.Int32	int
DeathsIndirect	9	System.Int32	int
DamageProperty	10	System.Int32	int
DamageCrops	11	System.Int32	int

ColumnName	ColumnOrdinal	DataType	ColumnType
Source	12	System.String	string
BeginLocation	13	System.String	string
EndLocation	14	System.String	string
BeginLat	15	System.Double	real
BeginLon	16	System.Double	real
EndLat	17	System.Double	real
EndLon	18	System.Double	real
EpisodeNarrative	19	System.String	string
EventNarrative	20	System.String	string
StormSummary	21	System.Object	dynamic