invoke operator

Article • 03/23/2023

Invokes a lambda expression that receives the source of `invoke` as a tabular argument.

> ⓘ Note
>
> For more information on how to declare lambda expressions that can accept tabular arguments, see **let statements**.

Syntax

T `| invoke` *function* `(` [*param1* `,` *param2*] `)`

Parameters

Name	Type	Required	Description
T	string	✓	The tabular source.
function	string	✓	The name of the lambda `let` expression or stored function name to be evaluated.
param1, *param2* ...	string		Any additional lambda arguments to pass to the function.

Returns

Returns the result of the evaluated expression.

Example

The following example shows how to use the `invoke` operator to call lambda `let` expression:

Run the query

```
Kusto
```

```
// clipped_average(): calculates percentiles limits, and then makes another
//                    pass over the data to calculate average with values
inside the percentiles
let clipped_average = (T:(x: long), lowPercentile:double,
upPercentile:double)
{
   let high = toscalar(T | summarize percentiles(x, upPercentile));
   let low = toscalar(T | summarize percentiles(x, lowPercentile));
   T
   | where x > low and x < high
   | summarize avg(x)
};
range x from 1 to 100 step 1
| invoke clipped_average(5, 99)
```

Output

avg_x
52

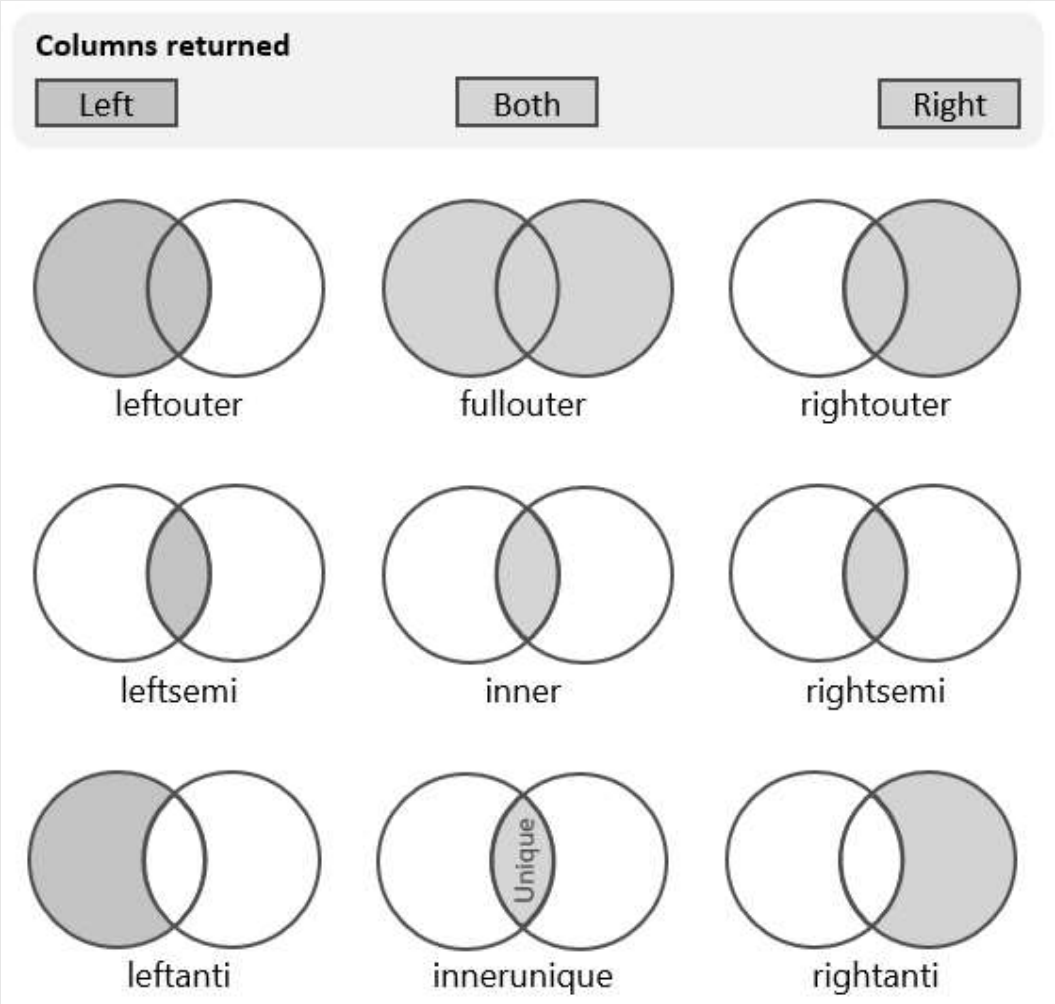
Feedback

join operator

Article • 06/19/2023

Merge the rows of two tables to form a new table by matching values of the specified columns from each table.

Kusto Query Language (KQL) offers many kinds of joins that each affect the schema and rows in the resultant table in different ways. For example, if you use an `inner` join, the table has the same columns as the left table, plus the columns from the right table. For best performance, if one table is always smaller than the other, use it as the left side of the `join` operator. The following image provides a visual representation of the operation performed by each join.



Syntax

LeftTable `|` `join` [`kind` `=` *JoinFlavor*] [*Hints*] `(` *RightTable* `)` `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
JoinFlavor	string		The type of join to perform: `innerunique`, `inner`, `leftouter`, `rightouter`, `fullouter`, `leftanti`, `rightanti`, `leftsemi`, `rightsemi`. The default is `innerunique`. For more information about join flavors, see Returns.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

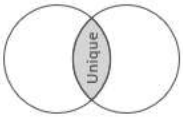
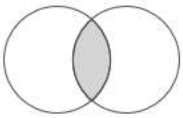
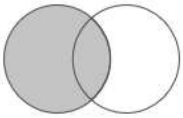
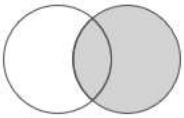
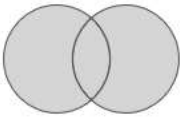
Hint key	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query

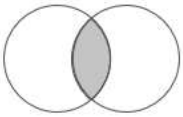
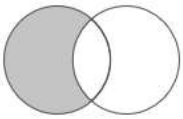
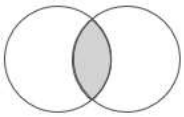
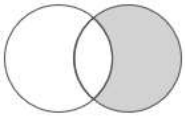
Hint key	Values	Description
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

> ⓘ Note
>
> The join hints don't change the semantic of `join` but may affect performance.

Returns

The return schema and rows depend on the join flavor. The join flavor is specified with the *kind* keyword. The following flavors of the join operator are supported:

Join flavor	Returns	Illustration
innerunique (default)	Inner join with left side deduplication **Schema**: All columns from both tables, including the matching keys **Rows**: All deduplicated rows from the left table that match rows from the right table	
inner	Standard inner join **Schema**: All columns from both tables, including the matching keys **Rows**: Only matching rows from both tables	
leftouter	Left outer join **Schema**: All columns from both tables, including the matching keys **Rows**: All records from the left table and only matching rows from the right table	
rightouter	Right outer join **Schema**: All columns from both tables, including the matching keys **Rows**: All records from the right table and only matching rows from the left table	
fullouter	Full outer join **Schema**: All columns from both tables, including the matching keys **Rows**: All records from both tables with unmatched cells populated with null	

Join flavor	Returns	Illustration
leftsemi	Left semi join **Schema**: All columns from the left table **Rows**: All records from the left table that match records from the right table	
leftanti, anti, leftantisemi	Left anti join and semi variant **Schema**: All columns from the left table **Rows**: All records from the left table that don't match records from the right table	
rightsemi	Right semi join **Schema**: All columns from the left table **Rows**: All records from the right table that match records from the left table	
rightanti, rightantisemi	Right anti join and semi variant **Schema**: All columns from the right table **Rows**: All records from the right table that don't match records from the left table	

Cross-join

KQL doesn't provide a cross-join flavor. However, you can achieve a cross-join effect by using a placeholder key approach.

In the following example, a placeholder key is added to both tables and then used for the inner join operation, effectively achieving a cross-join-like behavior:

```
X | extend placeholder=1 | join kind=inner (Y | extend placeholder=1) on placeholder
```

See also

- Cross-cluster join
- Broadcast join
- Shuffle query

Feedback

innerunique join

Article • 06/19/2023

The `innerunique` join flavor removes duplicate keys from the left side. This behavior ensures that the output contains a row for every combination of unique left and right keys.

By default, the `innerunique` join flavor is used if the `kind` parameter isn't specified. This default implementation is useful in log/trace analysis scenarios, where you aim to correlate two events based on a shared correlation ID. It allows you to retrieve all instances of the phenomenon while disregarding duplicate trace records that contribute to the correlation.

Syntax

LeftTable `|` `join` `kind=innerunique` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

Tip

For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from both tables, including the matching keys.
Rows: All deduplicated rows from the left table that match rows from the right table.

Examples

Use the default innerunique join

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
```

```
    'c',20,
    'c',30,
    'd',40
];
X | join Y on Key
```

Output

Key	Value1	Key1	Value2
b	2	b	10
c	4	c	20
c	4	c	30

> ⓘ Note
>
> The keys 'a' and 'd' don't appear in the output, since there were no matching keys on both left and right sides.

The query executed the default join, which is an inner join after deduplicating the left side based on the join key. The deduplication keeps only the first record. The resulting left side of the join after deduplication is:

Key	Value1
a	1
b	2
c	4

Two possible outputs from innerunique join

> ⓘ Note
>
> The `innerunique` join flavor may yield two possible outputs and both are correct. In the first output, the join operator randomly selected the first key that appears in t1, with the value "val1.1" and matched it with t2 keys. In the second output, the join operator randomly selected the second key that appears in t1, with the value "val1.2" and matched it with t2 keys.

Run the query

Kusto

```
let t1 = datatable(key: long, value: string)
    [
    1, "val1.1",
    1, "val1.2"
];
let t2 = datatable(key: long, value: string)
    [
    1, "val1.3",
    1, "val1.4"
];
t1
| join kind = innerunique
    t2
    on key
```

Output

key	value	key1	value1
1	val1.1	1	val1.3
1	val1.1	1	val1.4

Run the query

Kusto

```
let t1 = datatable(key: long, value: string)
    [
    1, "val1.1",
    1, "val1.2"
];
let t2 = datatable(key: long, value: string)
    [
    1, "val1.3",
    1, "val1.4"
];
t1
| join kind = innerunique
    t2
    on key
```

Output

key	value	key1	value1
1	val1.2	1	val1.3
1	val1.2	1	val1.4

- Kusto is optimized to push filters that come after the `join`, towards the appropriate join side, left or right, when possible.
- Sometimes, the flavor used is **innerunique** and the filter is propagated to the left side of the join. The flavor is automatically propagated and the keys that apply to that filter appear in the output.
- Use the previous example and add a filter `where value == "val1.2"`. It gives the second result and will never give the first result for the datasets:

Run the query

Kusto

```
let t1 = datatable(key: long, value: string)
    [
    1, "val1.1",
    1, "val1.2"
];
let t2 = datatable(key: long, value: string)
    [
    1, "val1.3",
    1, "val1.4"
];
t1
| join kind = innerunique
    t2
    on key
| where value == "val1.2"
```

Output

key	value	key1	value1
1	val1.2	1	val1.3
1	val1.2	1	val1.4

Get extended sign-in activities

Get extended activities from a `login` that some entries mark as the start and end of an activity.

Kusto

```
let Events = MyLogTable | where type=="Event" ;
Events
| where Name == "Start"
| project Name, City, ActivityId, StartTime=timestamp
| join (Events
    | where Name == "Stop"
        | project StopTime=timestamp, ActivityId)
    on ActivityId
| project City, ActivityId, StartTime, StopTime, Duration = StopTime -
StartTime
```

Kusto

```
let Events = MyLogTable | where type=="Event" ;
Events
| where Name == "Start"
| project Name, City, ActivityIdLeft = ActivityId, StartTime=timestamp
| join (Events
        | where Name == "Stop"
        | project StopTime=timestamp, ActivityIdRight = ActivityId)
    on $left.ActivityIdLeft == $right.ActivityIdRight
| project City, ActivityId, StartTime, StopTime, Duration = StopTime -
StartTime
```

See also

- Learn about other join flavors

Feedback

inner join

Article • 06/19/2023

The `inner` join flavor is like the standard inner join from the SQL world. An output record is produced whenever a record on the left side has the same join key as the record on the right side.

Syntax

LeftTable `|` `join` `kind=inner` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> Tip
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from both tables, including the matching keys.
Rows: Only matching rows from both tables.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'k',5,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40,
    'k',50
];
X | join kind=inner Y on Key
```

Output

Key	Value1	Key1	Value2
b	3	b	10
b	2	b	10
c	4	c	20
c	4	c	30
k	5	k	50

> ⓘ Note
>
> - (b,10) from the right side, was joined twice: with both (b,2) and (b,3) on the left.
> - (c,4) on the left side, was joined twice: with both (c,20) and (c,30) on the right.
> - (k,5) from the left and (k, 50) from the right was joined once.

See also

- Learn about other join flavors

Feedback

leftouter join

Article • 06/19/2023

The `leftouter` join flavor returns all the records from the left side table and only matching records from the right side table.

Syntax

LeftTable `|` `join` `kind=leftouter` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from both tables, including the matching keys.
Rows: All records from the left table and only matching rows from the right table.

Example

The result of a left outer join for tables X and Y always contains all records of the left table (X), even if the join condition doesn't find any matching record in the right table (Y).

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=leftouter Y on Key
```

Output

Key	Value1	Key1	Value2
a	1		
b	2	b	10
b	3	b	10
c	4	c	20
c	4	c	30

See also

- Learn about other join flavors

Feedback

rightouter join

Article • 06/19/2023

The `rightouter` join flavor returns all the records from the right side and only matching records from the left side. This join flavor resembles the rightouter join flavor, but the treatment of the tables is reversed.

Syntax

LeftTable `|` `join` `kind=rightouter` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from both tables, including the matching keys.
Rows: All records from the right table and only matching rows from the left table.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=rightouter Y on Key
```

Output

Key	Value1	Key1	Value2

Key	Value1	Key1	Value2
b	2	b	10
b	3	b	10
c	4	c	20
c	4	c	30
		d	40

See also

- Learn about other join flavors

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

fullouter join

Article • 06/19/2023

A `fullouter` join combines the effect of applying both left and right outer-joins. For columns of the table that lack a matching row, the result set contains `null` values. For those records that do match, a single row is produced in the result set containing fields populated from both tables.

Syntax

LeftTable `|` `join` `kind=fullouter` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from both tables, including the matching keys.
Rows: All records from both tables with unmatched cells populated with null.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=fullouter Y on Key
```

Output

Key	Value1	Key1	Value2

Key	Value1	Key1	Value2
a	1		
b	2	b	10
b	3	b	10
c	4	c	20
c	4	c	30
		d	40

See also

- Learn about other join flavors

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

leftsemi join

Article • 06/19/2023

The `leftsemi` join flavor returns all records from the left side that match a record from the right side. Only columns from the left side are returned.

Syntax

LeftTable `|` `join` `kind=leftsemi` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from the left table.
Rows: All records from the left table that match records from the right table.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=leftsemi Y on Key
```

Output

Key	Value1

Key	Value1
b	2
b	3
c	4

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

leftanti join

Article • 06/19/2023

The `leftanti` join flavor returns all records from the left side that don't match any record from the right side. The anti join models the "NOT IN" query.

> **Aliases**: `anti`, `leftantisemi`

Syntax

LeftTable `|` `join` `kind=leftanti` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from the left table.
Rows: All records from the left table that don't match records from the right table.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=leftanti Y on Key
```

Output

Key	Value1

Key	Value1
a	1

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

rightsemi join

Article • 06/29/2023

The `rightsemi` join flavor returns all records from the right side that match a record from the left side. Only columns from the right side are returned.

Syntax

LeftTable `|` `join` `kind=rightsemi` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from the right table.
Rows: All records from the right table that match records from the left table.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=rightsemi Y on Key
```

Output

Key	Value2

Key	Value2
b	10
c	20
c	30

Feedback

rightanti join

Article • 06/19/2023

The `rightanti` join flavor returns all records from the left side that don't match any record from the right side. The anti join models the "NOT IN" query.

> **Alias**: `rightantisemi`

Syntax

LeftTable `|` `join` `kind=rightanti` [*Hints*] *RightTable* `on` *Conditions*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The left table or tabular expression, sometimes called the outer table, whose rows are to be merged. Denoted as `$left`.
Hints	string		Zero or more space-separated join hints in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. For more information, see Hints.
RightTable	string	✓	The right table or tabular expression, sometimes called the inner table, whose rows are to be merged. Denoted as `$right`.
Conditions	string	✓	Determines how rows from *LeftTable* are matched with rows from *RightTable*. If the columns you want to match have the same name in both tables, use the syntax `ON` *ColumnName*. Otherwise, use the syntax `ON $left.`*LeftColumn* `==` `$right.`*RightColumn. To specify multiple conditions, you can either use the "and" keyword or separate them with commas. If you use commas, the conditions are evaluated using the "and" logical operator.

> **Tip**
>
> For best performance, if one table is always smaller than the other, use it as the left side of the join.

Hints

Parameters name	Values	Description
`hint.remote`	`auto`, `left`, `local`, `right`	See Cross-Cluster Join
`hint.strategy=broadcast`	Specifies the way to share the query load on cluster nodes.	See broadcast join
`hint.shufflekey=<key>`	The `shufflekey` query shares the query load on cluster nodes, using a key to partition data.	See shuffle query
`hint.strategy=shuffle`	The `shuffle` strategy query shares the query load on cluster nodes, where each node processes one partition of the data.	See shuffle query

Returns

Schema: All columns from the right table.
Rows: All records from the right table that don't match records from the left table.

Example

Run the query

Kusto

```
let X = datatable(Key:string, Value1:long)
[
    'a',1,
    'b',2,
    'b',3,
    'c',4
];
let Y = datatable(Key:string, Value2:long)
[
    'b',10,
    'c',20,
    'c',30,
    'd',40
];
X | join kind=rightanti Y on Key
```

Output

Key	Value1

Key	Value1
a	1

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Cross-cluster join

Article • 06/27/2023

For general discussion on cross-cluster queries, see cross-cluster or cross-database queries

It's possible to do join operation on datasets residing on different clusters. For example:

Kusto

```
T | ... | join (cluster("SomeCluster").database("SomeDB").T2 | ...) on Col1
// (1)

cluster("SomeCluster").database("SomeDB").T | ... | join
(cluster("SomeCluster2").database("SomeDB2").T2 | ...) on Col1 // (2)
```

In the example above, the join operation is a cross-cluster join, assuming that current cluster isn't "SomeCluster" or "SomeCluster2".

In the following example:

Kusto

```
cluster("SomeCluster").database("SomeDB").T | ... | join
(cluster("SomeCluster").database("SomeDB2").T2 | ...) on Col1
```

the join operation isn't a cross-cluster join because both its operands originate on the same cluster.

When Kusto encounters a cross-cluster join, it will automatically decide where to execute the join operation itself. This decision can have one of the three possible outcomes:

- Execute join operation on the cluster of the left operand. The right operand is first fetched by this cluster. (join in example **(1)** will be executed on the local cluster)
- Execute join operation on the cluster of the right operand. The left operand is first fetched by this cluster. (join in example **(2)** will be executed on the "SomeCluster2")
- Execute join operation locally (meaning on the cluster that received the query). Both operands are first fetched by the local cluster.

The actual decision depends on the specific query. The automatic join remoting strategy is (simplified version): "If one of the operands is local, join will be executed locally. If both operands are remote, join will be executed on the cluster of the right operand".

Sometimes the performance of the query can be improved if automatic remoting strategy isn't followed. In this case, execute join operation on the cluster of the largest operand.

"Example 1" is set to run on the local cluster, but if the dataset produced by `T | ...` is smaller than one produced by `cluster("SomeCluster").database("SomeDB").T2 | ...` then it would be more efficient to execute the join operation on `SomeCluster` instead of on the local cluster.

To execute the join on `SomeCluster`, specify the remote strategy as `right`. Then, the cluster will execute the join on the right cluster even if the left cluster is the local cluster.

Kusto

```
T | ... | join hint.remote=right (cluster("SomeCluster").database("DB").T2 |
...) on Col1
```

Following are legal values for `strategy`:

- `left` - execute join on the cluster of the left operand
- `right` - execute join on the cluster of the right operand
- `local` - execute join on the cluster of the current cluster
- `auto` - (default) let Kusto make the automatic remoting decision

> **Note**
>
> The join remoting hint will be ignored by Kusto if the hinted strategy isn't applicable to the join operation.

Feedback

Broadcast join

Article • 04/11/2023

Today, regular joins are executed on a single cluster node. Broadcast join is an execution strategy of join that distributes the join over cluster nodes. This strategy is useful when the left side of the join is small (up to several tens of MBs). In this case, a broadcast join is more performant than a regular join.

> **Note**
>
> If the left side of the join is larger than several tens of MBs, the query will fail.
>
> You can run the following query to estimate the size of the left side, in bytes:

Kusto

```
leftSide
| summarize sum(estimate_data_size(*))
```

If left side of the join is a small dataset, then you may run join in broadcast mode using the following syntax (hint.strategy = broadcast):

Kusto

```
leftSide
| join hint.strategy = broadcast (factTable) on key
```

The performance improvement is more noticeable in scenarios where the join is followed by other operators such as `summarize`. See the following query for example:

Kusto

```
leftSide
| join hint.strategy = broadcast (factTable) on Key
| summarize dcount(Messages) by Timestamp, Key
```

Feedback

Time window join

Article • 04/11/2023

It's often useful to join between two large data sets on some high-cardinality key, such as an operation ID or a session ID, and further limit the right-hand-side ($right) records that need to match up with each left-hand-side ($left) record by adding a restriction on the "time-distance" between datetime columns on the left and on the right.

The function is useful in a join, like in the following scenario:

- Join between two large data sets according to some high-cardinality key, such as an operation ID or a session ID.
- Limit the right-hand-side ($right) records that need to match up with each left-hand-side ($left) record, by adding a restriction on the "time-distance" between datetime columns on the left and on the right.

The above operation differs from the usual Kusto join operation, since for the *`equi-join`* part of matching the high-cardinality key between the left and right data sets, the system can also apply a distance function and use it to considerably speed up the join.

> ⓘ **Note**
>
> A distance function doesn't behave like equality (that is, when both dist(x,y) and dist(y,z) are true it doesn't follow that dist(x,z) is also true.) Internally, we sometimes refer to this as "diagonal join".

For example, if you want to identify event sequences within a relatively small time window, assume that you have a table `T` with the following schema:

- `SessionId`: A column of type `string` with correlation IDs.
- `EventType`: A column of type `string` that identifies the event type of the record.
- `Timestamp`: A column of type `datetime` indicates when the event described by the record happened.

Run the query

Kusto

```
let T = datatable(SessionId:string, EventType:string, Timestamp:datetime)
[
    '0', 'A', datetime(2017-10-01 00:00:00),
    '0', 'B', datetime(2017-10-01 00:01:00),
```

```
    '1', 'B', datetime(2017-10-01 00:02:00),
    '1', 'A', datetime(2017-10-01 00:03:00),
    '3', 'A', datetime(2017-10-01 00:04:00),
    '3', 'B', datetime(2017-10-01 00:10:00),
];
T
```

Output

SessionId	EventType	Timestamp
0	A	2017-10-01 00:00:00.0000000
0	B	2017-10-01 00:01:00.0000000
1	B	2017-10-01 00:02:00.0000000
1	A	2017-10-01 00:03:00.0000000
3	A	2017-10-01 00:04:00.0000000
3	B	2017-10-01 00:10:00.0000000

Problem statement

Our query should answer the following question:

Find all the session IDs in which event type `A` was followed by an event type `B` within a `1min` time window.

> ⓘ Note
>
> In the sample data above, the only such session ID is `0`.

Semantically, the following query answers this question, albeit inefficiently.

Run the query

Kusto

```
T
| where EventType == 'A'
| project SessionId, Start=Timestamp
| join kind=inner
    (
    T
    | where EventType == 'B'
    | project SessionId, End=Timestamp
```

```
    ) on SessionId
| where (End - Start) between (0min .. 1min)
| project SessionId, Start, End
```

Output

SessionId	Start	End
0	2017-10-01 00:00:00.0000000	2017-10-01 00:01:00.0000000

To optimize this query, we can rewrite it as described below so that the time window is expressed as a join key.

Rewrite the query to account for the time window

Rewrite the query so that the `datetime` values are "discretized" into buckets whose size is half the size of the time window. Use Kusto's *`equi-join`* to compare those bucket IDs.

Kusto

```
let lookupWindow = 1min;
let lookupBin = lookupWindow / 2.0; // lookup bin = equal to 1/2 of the
lookup window
T
| where EventType == 'A'
| project SessionId, Start=Timestamp,
          // TimeKey on the left side of the join is mapped to a discrete
time axis for the join purpose
          TimeKey = bin(Timestamp, lookupBin)
| join kind=inner
    (
    T
    | where EventType == 'B'
    | project SessionId, End=Timestamp,
              // TimeKey on the right side of the join - emulates event 'B'
appearing several times
              // as if it was 'replicated'
              TimeKey = range(bin(Timestamp-lookupWindow, lookupBin),
                              bin(Timestamp, lookupBin),
                              lookupBin)
    // 'mv-expand' translates the TimeKey array range into a column
    | mv-expand TimeKey to typeof(datetime)
    ) on SessionId, TimeKey
| where (End - Start) between (0min .. lookupWindow)
| project SessionId, Start, End
```

Runnable query reference (with table inlined)

Run the query

Kusto

```
let T = datatable(SessionId:string, EventType:string, Timestamp:datetime)
[
    '0', 'A', datetime(2017-10-01 00:00:00),
    '0', 'B', datetime(2017-10-01 00:01:00),
    '1', 'B', datetime(2017-10-01 00:02:00),
    '1', 'A', datetime(2017-10-01 00:03:00),
    '3', 'A', datetime(2017-10-01 00:04:00),
    '3', 'B', datetime(2017-10-01 00:10:00),
];
let lookupWindow = 1min;
let lookupBin = lookupWindow / 2.0;
T
| where EventType == 'A'
| project SessionId, Start=Timestamp, TimeKey = bin(Timestamp, lookupBin)
| join kind=inner
    (
    T
    | where EventType == 'B'
    | project SessionId, End=Timestamp,
              TimeKey = range(bin(Timestamp-lookupWindow, lookupBin),
                              bin(Timestamp, lookupBin),
                              lookupBin)
    | mv-expand TimeKey to typeof(datetime)
    ) on SessionId, TimeKey
| where (End - Start) between (0min .. lookupWindow)
| project SessionId, Start, End
```

Output

SessionId	Start	End
0	2017-10-01 00:00:00.0000000	2017-10-01 00:01:00.0000000

5M data query

The next query emulates a data set of 5M records and ~1M IDs and runs the query with the technique described above.

Run the query

Kusto

```
let T = range x from 1 to 5000000 step 1
| extend SessionId = rand(1000000), EventType = rand(3), Time=datetime(2017-
01-01)+(x * 10ms)
| extend EventType = case(EventType < 1, "A",
                          EventType < 2, "B",
                          "C");
```

```
let lookupWindow = 1min;
let lookupBin = lookupWindow / 2.0;
T
| where EventType == 'A'
| project SessionId, Start=Time, TimeKey = bin(Time, lookupBin)
| join kind=inner
    (
    T
    | where EventType == 'B'
    | project SessionId, End=Time,
              TimeKey = range(bin(Time-lookupWindow, lookupBin),
                              bin(Time, lookupBin),
                              lookupBin)
    | mv-expand TimeKey to typeof(datetime)
    ) on SessionId, TimeKey
| where (End - Start) between (0min .. lookupWindow)
| project SessionId, Start, End
| count
```

Output

Count
3344

Feedback

lookup operator

Article • 03/12/2023

Extends the columns of a fact table with values looked-up in a dimension table.

Kusto

```
FactTable | lookup kind=leftouter (DimensionTable) on CommonColumn,
$left.Col1 == $right.Col2
```

Here, the result is a table that extends the `FactTable` (`$left`) with data from `DimensionTable` (referenced by `$right`) by performing a lookup of each pair (`CommonColumn`,`Col`) from the former table with each pair (`CommonColumn1`,`Col2`) in the latter table. For the differences between fact and dimension tables, see fact and dimension tables.

The `lookup` operator performs an operation similar to the join operator with the following differences:

- The result doesn't repeat columns from the `$right` table that are the basis for the join operation.
- Only two kinds of lookup are supported, `leftouter` and `inner`, with `leftouter` being the default.
- In terms of performance, the system by default assumes that the `$left` table is the larger (facts) table, and the `$right` table is the smaller (dimensions) table. This is exactly opposite to the assumption used by the `join` operator.
- The `lookup` operator automatically broadcasts the `$right` table to the `$left` table (essentially, behaves as if `hint.broadcast` was specified). This limits the size of the `$right` table.

> ⓘ **Note**
>
> If the right side of the lookup is larger than several tens of MBs, the query will fail.
>
> You can run the following query to estimate the size of the right side in bytes:
>
> Kusto
>
> ```
> rightSide
> | summarize sum(estimate_data_size(*))
> ```

Syntax

LeftTable `|` `lookup` [`kind` `=` (`leftouter`|`inner`)] `(`*RightTable*`)` `on` *Attributes*

Parameters

Name	Type	Required	Description
LeftTable	string	✓	The table or tabular expression that is the basis for the lookup. Denoted as `$left`.
RightTable	string	✓	The table or tabular expression that is used to "populate" new columns in the fact table. Denoted as `$right`.
Attributes	string	✓	A comma-delimited list of one or more rules that describe how rows from *LeftTable* are matched to rows from *RightTable*. Multiple rules are evaluated using the `and` logical operator. See Rules.
`kind`	string		Determines how to treat rows in *LeftTable* that have no match in *RightTable*. By default, `leftouter` is used, which means all those rows will appear in the output with null values used for the missing values of *RightTable* columns added by the operator. If `inner` is used, such rows are omitted from the output. Other kinds of join aren't supported by the `lookup` operator.

Rules

Rule kind	Syntax	Predicate
Equality by name	*ColumnName*	`where` *LeftTable.ColumnName* `==` *RightTable.ColumnName*
Equality by value	`$left.`*LeftColumn* `==` `$right.`*RightColumn*	`where` `$left.`*LeftColumn* `==` `$right.`*RightColumn

> ⓘ Note
>
> In case of 'equality by value', the column names *must* be qualified with the applicable owner table denoted by `$left` and `$right` notations.

Returns

A table with:

- A column for every column in each of the two tables, including the matching keys. The columns of the right side will be automatically renamed if there are name conflicts.
- A row for every match between the input tables. A match is a row selected from one table that has the same value for all the `on` fields as a row in the other table.
- The Attributes (lookup keys) will appear only once in the output table.
- If `kind` is unspecified or `kind=leftouter`, then in addition to the inner matches, there's a row for every row on the left (and/or right), even if it has no match. In that case, the unmatched output cells contain nulls.
- If `kind=inner`, then there's a row in the output for every combination of matching rows from left and right.

Examples

Run the query

Kusto

```
let FactTable=datatable(Row:string,Personal:string,Family:string) [
  "1", "Bill",   "Gates",
  "2", "Bill",   "Clinton",
  "3", "Bill",   "Clinton",
  "4", "Steve",  "Ballmer",
  "5", "Tim",    "Cook"
];
let DimTable=datatable(Personal:string,Family:string,Alias:string) [
  "Bill",  "Gates",   "billg",
  "Bill",  "Clinton", "billc",
  "Steve", "Ballmer", "steveb",
  "Tim",   "Cook",    "timc"
];
FactTable
| lookup kind=leftouter DimTable on Personal, Family
```

Row	Personal	Family	Alias
1	Bill	Gates	billg
2	Bill	Clinton	billc
3	Bill	Clinton	billc
4	Steve	Ballmer	steveb

Row	Personal	Family	Alias
5	Tim	Cook	timc

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

mv-apply operator

Article • 05/30/2023

Applies a subquery to each record, and returns the union of the results of all subqueries.

For example, assume a table `T` has a column `Metric` of type `dynamic` whose values are arrays of `real` numbers. The following query locates the two biggest values in each `Metric` value, and return the records corresponding to these values.

Kusto

```
T | mv-apply Metric to typeof(real) on
(
   top 2 by Metric desc
)
```

The `mv-apply` operator has the following processing steps:

1. Uses the mv-expand operator to expand each record in the input into subtables (order is preserved).
2. Applies the subquery for each of the subtables.
3. Adds zero or more columns to the resulting subtable. These columns contain the values of the source columns that aren't expanded, and are repeated where needed.
4. Returns the union of the results.

The `mv-apply` operator gets the following inputs:

1. One or more expressions that evaluate into dynamic arrays to expand. The number of records in each expanded subtable is the maximum length of each of those dynamic arrays. Null values are added where multiple expressions are specified and the corresponding arrays have different lengths.

2. Optionally, the names to assign the values of the expressions after expansion. These names become the columns names in the subtables. If not specified, the original name of the column is used when the expression is a column reference. A random name is used otherwise.

   > ⓘ **Note**
   >
   > It is recommended to use the default column names.

3. The data types of the elements of those dynamic arrays, after expansion. These become the column types of the columns in the subtables. If not specified, `dynamic` is used.

4. Optionally, the name of a column to add to the subtables that specifies the 0-based index of the element in the array that resulted in the subtable record.

5. Optionally, the maximum number of array elements to expand.

The `mv-apply` operator can be thought of as a generalization of the mv-expand operator (in fact, the latter can be implemented by the former, if the subquery includes only projections.)

Syntax

T `|` `mv-apply` [*ItemIndex*] *ColumnsToExpand* [*RowLimit*] `on` `(` *SubQuery* `)`

Where *ItemIndex* has the syntax:

`with_itemindex` `=` *IndexColumnName*

ColumnsToExpand is a comma-separated list of one or more elements of the form:

[*Name* `=`] *ArrayExpression* [`to` `typeof` `(`*Typename*`)`]

RowLimit is simply:

`limit` *RowLimit*

and *SubQuery* has the same syntax of any query statement.

Parameters

Name	Type	Required	Description
ItemIndex	string		Indicates the name of a column of type `long` that's appended to the input as part of the array-expansion phase and indicates the 0-based array index of the expanded value.
Name	string		The name to assign the array-expanded values of each array-expanded expression. If not specified, the name of the column is used if available. A random name is generated if *ArrayExpression* isn't a simple column name.

Name	Type	Required	Description
ArrayExpression	dynamic	✓	The array whose values are array-expanded. If the expression is the name of a column in the input, the input column is removed from the input and a new column of the same name, or *ColumnName* if specified, appears in the output.
Typename	string		The name of the type that the individual elements of the `dynamic` array *ArrayExpression* take. Elements that don't conform to this type are replaced by a null value. If unspecified, `dynamic` is used by default.
RowLimit	int		A limit on the number of records to generate from each record of the input. If unspecified, 2147483647 is used.
SubQuery	string		A tabular query expression with an implicit tabular source that gets applied to each array-expanded subtable.

> ⓘ Note
>
> Unlike the **mv-expand** operator, the `mv-apply` operator doesn't support `bagexpand=array` expansion. If the expression to be expanded is a property bag and not an array, you can use an inner `mv-expand` operator (see example below).

Examples

Getting the largest element from the array

Run the query

Kusto

```
let _data =
    range x from 1 to 8 step 1
    | summarize l=make_list(x) by xMod2 = x % 2;
_data
| mv-apply element=l to typeof(long) on
    (
    top 1 by element
    )
```

Output

xMod2	l	element
1	[1, 3, 5, 7]	7
0	[2, 4, 6, 8]	8

Calculating the sum of the largest two elements in an array

Run the query

Kusto

```
let _data =
    range x from 1 to 8 step 1
    | summarize l=make_list(x) by xMod2 = x % 2;
_data
| mv-apply l to typeof(long) on
    (
    top 2 by l
    | summarize SumOfTop2=sum(l)
    )
```

Output

xMod2	l	SumOfTop2
1	[1,3,5,7]	12
0	[2,4,6,8]	14

Select elements in arrays

Run the query

Kusto

```
datatable (Val:int, Arr1:dynamic, Arr2:dynamic)
[ 1, dynamic(['A1', 'A2', 'A3']),       dynamic([10, 30, 7]),
  7, dynamic(['B1', 'B2', 'B5']),       dynamic([15, 11, 50]),
  3, dynamic(['C1', 'C2', 'C3', 'C4']), dynamic([6, 40, 20, 8])
]
| mv-apply NewArr1=Arr1, NewArr2=Arr2 to typeof(long) on (
 top 2 by NewArr2
 | summarize NewArr1=make_list(NewArr1), NewArr2=make_list(NewArr2)
)
```

Output

Val1	Arr1	Arr2	`NewArr1`	`NewArr2`
1	["A1","A2","A3"]	[10,30,7]	["A2","A1"]	[30,10]
7	["B1","B2","B5"]	[15,11,50]	["B5","B1"]	[50,15]
3	["C1","C2","C3","C4"]	[6,40,20,8]	["C2","C3"]	[40,20]

Using `with_itemindex` for working with a subset of the array

Run the query

Kusto

```
let _data =
    range x from 1 to 10 step 1
    | summarize l=make_list(x) by xMod2 = x % 2;
_data
| mv-apply with_itemindex=index element=l to typeof(long) on
    (
    // here you have 'index' column
    where index >= 3
    )
| project index, element
```

Output

index	element
3	7
4	9
3	8
4	10

Using mutiple columns to join element of 2 arrays

Run the query

Kusto

```
datatable (Val: int, Arr1: dynamic, Arr2: dynamic)
[
    1, dynamic(['A1', 'A2', 'A3']), dynamic(['B1', 'B2', 'B3']),
    5, dynamic(['C1', 'C2']), dynamic(['D1', 'D2'])
]
| mv-apply Arr1, Arr2 on (
    extend Out = strcat(Arr1, "_", Arr2)
    | summarize Arr1 = make_list(Arr1), Arr2 = make_list(Arr2), Out=
make_list(Out)
    )
```

Output

Val	Arr1	Arr2	`Out`
1	["A1","A2","A3"]	["B1","B2","B3"]	["A1_B1","A2_B2","A3_B3"]
5	["C1","C2"]	["D1","D2"]	["C1_D1","C2_D2"]

Applying mv-apply to a property bag

In the following example, `mv-apply` is used in combination with an inner `mv-expand` to remove values that don't start with "555" from a property bag:

Run the query

Kusto

```
datatable(SourceNumber: string, TargetNumber: string, CharsCount: long)
[
    '555-555-1234', '555-555-1212', 46,
    '555-555-1212', '', int(null)
]
| extend values = pack_all()
| mv-apply removeProperties = values on
    (
    mv-expand kind = array values
    | where values[1] !startswith "555"
    | summarize propsToRemove = make_set(values[0])
    )
| extend values = bag_remove_keys(values, propsToRemove)
| project-away propsToRemove
```

Output

SourceNumber	TargetNumber	CharsCount	values

SourceNumber	TargetNumber	CharsCount	values
555-555-1234	555-555-1212	46	{ "SourceNumber": "555-555-1234", "TargetNumber": "555-555-1212" }
555-555-1212			{ "SourceNumber": "555-555-1212" }

See also

- mv-expand operator

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

mv-expand operator

Article • 03/19/2023

Expands multi-value dynamic arrays or property bags into multiple records.

`mv-expand` can be described as the opposite of the aggregation operators that pack multiple values into a single dynamic-typed array or property bag, such as `summarize` ... `make-list()` and `make-series`. Each element in the (scalar) array or property bag generates a new record in the output of the operator. All columns of the input that aren't expanded are duplicated to all the records in the output.

Syntax

T `|mv-expand` [`bagexpansion=`(`bag` | `array`)] [`with_itemindex=` *IndexColumnName*] *ColumnName* [`to typeof(` *Typename*`)`] [`,` *ColumnName* ...] [`limit` *Rowlimit*]

T `|mv-expand` [`bagexpansion=`(`bag` | `array`)] [*Name* `=`] *ArrayExpression* [`to typeof(`*Typename*`)`] [`,` [*Name* `=`] *ArrayExpression* [`to typeof(`*Typename*`)`] ...] [`limit` *Rowlimit*]

Parameters

Name	Type	Required	Description
ColumnName, *ArrayExpression*	string	✓	A column reference, or a scalar expression with a value of type `dynamic` that holds an array or a property bag. The individual top-level elements of the array or property bag get expanded into multiple records. When *ArrayExpression* is used and *Name* doesn't equal any input column name, the expanded value is extended into a new column in the output. Otherwise, the existing *ColumnName* is replaced.
Name	string		A name for the new column.
Typename	string	✓	Indicates the underlying type of the array's elements, which becomes the type of the column produced by the `mv-expand` operator. The operation of applying type is cast-only and doesn't include parsing or type-conversion. Array elements that don't conform with the declared type become `null` values.

Name	Type	Required	Description
RowLimit	int		The maximum number of rows generated from each original row. The default is 2147483647. `mvexpand` is a legacy and obsolete form of the operator `mv-expand`. The legacy version has a default row limit of 128.
IndexColumnName	string		If `with_itemindex` is specified, the output includes another column named *IndexColumnName* that contains the index starting at 0 of the item in the original expanded collection.

Returns

For each record in the input, the operator returns zero, one, or many records in the output, as determined in the following way:

1. Input columns that aren't expanded appear in the output with their original value. If a single input record is expanded into multiple output records, the value is duplicated to all records.

2. For each *ColumnName* or *ArrayExpression* that is expanded, the number of output records is determined for each value as explained in modes of expansion. For each input record, the maximum number of output records is calculated. All arrays or property bags are expanded "in parallel" so that missing values (if any) are replaced by null values. Elements are expanded into rows in the order that they appear in the original array/bag.

3. If the dynamic value is null, then a single record is produced for that value (null). If the dynamic value is an empty array or property bag, no record is produced for that value. Otherwise, as many records are produced as there are elements in the dynamic value.

The expanded columns are of type `dynamic`, unless they're explicitly typed by using the `to typeof()` clause.

Modes of expansion

Two modes of property bag expansions are supported:

- `bagexpansion=bag` or `kind=bag`: Property bags are expanded into single-entry property bags. This mode is the default mode.

- `bagexpansion=array` or `kind=array`: Property bags are expanded into two-element `[`*key*`,`*value*`]` array structures, allowing uniform access to keys and values. This mode also allows, for example, running a distinct-count aggregation over property names.

Examples

Single column - array expansion

Run the query

Kusto

```
datatable (a: int, b: dynamic)
[
    1, dynamic([10, 20]),
    2, dynamic(['a', 'b'])
]
| mv-expand b
```

Output

a	b
1	10
1	20
2	a
2	b

Single column - bag expansion

A simple expansion of a single column:

Run the query

Kusto

```
datatable (a: int, b: dynamic)
[
    1, dynamic({"prop1": "a1", "prop2": "b1"}),
    2, dynamic({"prop1": "a2", "prop2": "b2"})
```

```
]
| mv-expand b
```

Output

a	b
1	{"prop1": "a1"}
1	{"prop2": "b1"}
2	{"prop1": "a2"}
2	{"prop2": "b2"}

Single column - bag expansion to key-value pairs

A simple bag expansion to key-value pairs:

Run the query

Kusto

```
datatable (a: int, b: dynamic)
[
    1, dynamic({"prop1": "a1", "prop2": "b1"}),
    2, dynamic({"prop1": "a2", "prop2": "b2"})
]
| mv-expand bagexpansion=array b
| extend key = b[0], val=b[1]
```

Output

a	b	key	val
1	["prop1","a1"]	prop1	a1
1	["prop2","b1"]	prop2	b1
2	["prop1","a2"]	prop1	a2
2	["prop2","b2"]	prop2	b2

Zipped two columns

Expanding two columns will first 'zip' the applicable columns and then expand them:

Run the query

Kusto

```
datatable (a: int, b: dynamic, c: dynamic)[
    1, dynamic({"prop1": "a", "prop2": "b"}), dynamic([5, 4, 3])
]
| mv-expand b, c
```

Output

a	b	c
1	{"prop1":"a"}	5
1	{"prop2":"b"}	4
1		3

Cartesian product of two columns

If you want to get a Cartesian product of expanding two columns, expand one after the other:

Run the query

Kusto

```
datatable (a: int, b: dynamic, c: dynamic)
[
    1, dynamic({"prop1": "a", "prop2": "b"}), dynamic([5, 6])
]
| mv-expand b
| mv-expand c
```

Output

a	b	c
1	{ "prop1": "a"}	5
1	{ "prop1": "a"}	6
1	{ "prop2": "b"}	5
1	{ "prop2": "b"}	6

Convert output

To force the output of an mv-expand to a certain type (default is dynamic), use `to typeof`:

Run the query

Kusto

```
datatable (a: string, b: dynamic, c: dynamic)[
    "Constant", dynamic([1, 2, 3, 4]), dynamic([6, 7, 8, 9])
]
| mv-expand b, c to typeof(int)
| getschema
```

Output

ColumnName	ColumnOrdinal	DateType	ColumnType
a	0	System.String	string
b	1	System.Object	dynamic
c	2	System.Int32	int

Notice column `b` is returned as `dynamic` while `c` is returned as `int`.

Using with_itemindex

Expansion of an array with `with_itemindex`:

Run the query

Kusto

```
range x from 1 to 4 step 1
| summarize x = make_list(x)
| mv-expand with_itemindex=Index x
```

Output

x	Index
1	0

x	Index
2	1
3	2
4	3

See also

- For more examples, see Chart count of live activities over time.
- mv-apply operator.
- For the opposite of the mv-expand operator, see summarize make_list().
- For expanding dynamic JSON objects into columns using property bag keys, see bag_unpack() plugin.

Feedback

sort operator

Article • 01/16/2023

Sorts the rows of the input table into order by one or more columns.

> The `sort` and `order` operators are equivalent

Syntax

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

Parameters

Name	Type	Required	Description
T	string	✓	The tabular input to sort.
column	scalar	✓	The column of *T* by which to sort. The type of the column values must be numeric, date, time or string.
`asc` or `desc`	string		`asc` sorts into ascending order, low to high. Default is `desc`, high to low.
`nulls first` or `nulls last`	string		`nulls first` will place the null values at the beginning and `nulls last` will place the null values at the end. Default for `asc` is `nulls first`. Default for `desc` is `nulls last`.

Returns

A copy of the input table sorted in either ascending or descending order based on the provided column.

Example

The following example shows storm events by state in alphabetical order with the most recent storms in each state appearing first.

Run the query

```
Kusto
```