# scan operator

Article • 01/31/2023

Scans data, matches, and builds sequences based on the predicates.

Matching records are determined according to predicates defined in the operator's steps. A predicate can depend on the state that is generated by previous steps. The output for the matching record is determined by the input record and assignments defined in the operator's steps.

Steps are evaluated from last to first, according to the scan logic.

Kusto

```
T
| sort by Timestamp asc
| scan with
(
    step s1 output=last: Event == "Start";
    step s2: Event != "Start" and Event != "Stop" and Timestamp -
s1.Timestamp <= 5m;
    step s3: Event == "Stop"  and Ts - s1.Timestamp <= 5m;
)
```

## Syntax

*T* `| scan` [ `with_match_id` `=` *MatchIdColumnName* ] [ `declare` `(` *ColumnDeclarations* `)` ] `with` `(` *StepDefinitions* `)`

### *ColumnDeclarations* syntax

*ColumnName* `:` *ColumnType*[`=` *DefaultValue* ] [`,` ... ]

### *StepDefinition* syntax

`step` *StepName* [ `output` = `all` | `last` | `none`] `:` *Condition* [ `=>` *Column* `=` *Assignment* [`,` ... ] ] `;`

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|

| Name | Type | Required | Description |
|---|---|---|---|
| *MatchIdColumnName* | string | | The name of a column of type `long` that is appended to the output as part of the scan execution. Indicates the 0-based index of the match for the row. |
| *ColumnDeclarations* | string | | Declares an extension to the schema of the operator's source. Additional columns are assigned in the steps or *DefaultValue* if not assigned. *DefaultValue* is `null` if not specified. |
| *StepName* | string | ✓ | Used to reference values in the state of scan for conditions and assignments. The step name must be unique. |
| *Condition* | string | ✓ | An expression that evaluates to a `bool`, `true` or `false`, that defines which records from the input matches the step. A record matches the step when the condition is `true` with the step's state or with the previous step's state. |
| *Assignment* | string | | A scalar expression that is assigned to the corresponding column when a record matches a step. |
| `output` | string | | Controls the output logic of the step on repeated matches. `all` outputs all records matching the step, `last` outputs only the last record in a series of repeating matches for the step, and `none` does not output records matching the step. The default is `all`. |

# Returns

A record for each match of a record from the input to a step. The schema of the output is the schema of the source extended with the column in the `declare` clause.

# Examples

## Cumulative sum

Calculate the cumulative sum for an input column. The result of this example is equivalent to using row_cumsum().

Run the query

Kusto

```
range x from 1 to 5 step 1
| scan declare (cumulative_x:long=0) with
(
    step s1: true => cumulative_x = x + s1.cumulative_x;
)
```

**Output**

| x | cumulative_x |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 15 |

## Cumulative sum on multiple columns with a reset condition

Calculate the cumulative sum for two input column, reset the sum value to the current row value whenever the cumulative sum reached 10 or more.

Run the query

Kusto

```
range x from 1 to 5 step 1
| extend y = 2 * x
| scan declare (cumulative_x:long=0, cumulative_y:long=0) with
(
    step s1: true => cumulative_x = iff(s1.cumulative_x >= 10, x, x +
s1.cumulative_x),
                     cumulative_y = iff(s1.cumulative_y >= 10, y, y +
s1.cumulative_y);
)
```

**Output**

| x | y | cumulative_x | cumulative_y |
|---|---|---|---|
| 1 | 2 | 1 | 2 |

| x | y | cumulative_x | cumulative_y |
|---|---|---|---|
| 2 | 4 | 3 | 6 |
| 3 | 6 | 6 | 12 |
| 4 | 8 | 10 | 8 |
| 5 | 10 | 5 | 18 |

## Fill forward a column

Fill forward a string column. Each empty value is assigned the last seen non-empty value.

Run the query

Kusto

```
let Events = datatable (Ts: timespan, Event: string) [
    0m, "A",
    1m, "",
    2m, "B",
    3m, "",
    4m, "",
    6m, "C",
    8m, "",
    11m, "D",
    12m, ""
]
;
Events
| sort by Ts asc
| scan declare (Event_filled: string="") with
(
    step s1: true => Event_filled = iff(isempty(Event), s1.Event_filled,
Event);
)
```

### Output

| Ts | Event | Event_filled |
|---|---|---|
| 00:00:00 | A | A |
| 00:01:00 | | A |
| 00:02:00 | B | B |

| Ts | Event | Event_filled |
|---|---|---|
| 00:03:00 | | B |
| 00:04:00 | | B |
| 00:06:00 | C | C |
| 00:08:00 | | C |
| 00:11:00 | D | D |
| 00:12:00 | | D |

## Sessions tagging

Divide the input into sessions: a session ends 30 minutes after the first event of the session, after which a new session starts. Note the use of `with_match_id` flag which assigns a unique value for each distinct match (session) of *scan*. Also note the special use of two *steps* in this example, `inSession` has `true` as condition so it captures and outputs all the records from the input while `endSession` captures records that happen more than 30m from the `sessionStart` value for the current match. The `endSession` step has `output=none` meaning it doesn't produce output records. The `endSession` step is used to advance the state of the current match from `inSession` to `endSession`, allowing a new match (session) to begin, starting from the current record.

Run the query

Kusto

```
let Events = datatable (Ts: timespan, Event: string) [
    0m, "A",
    1m, "A",
    2m, "B",
    3m, "D",
    32m, "B",
    36m, "C",
    38m, "D",
    41m, "E",
    75m, "A"
]
;
Events
| sort by Ts asc
| scan with_match_id=session_id declare (sessionStart: timespan) with
(
    step inSession: true => sessionStart =
iff(isnull(inSession.sessionStart), Ts, inSession.sessionStart);
```

```
    step endSession output=none: Ts - inSession.sessionStart > 30m;
)
```

**Output**

| Ts | Event | sessionStart | session_id |
|---|---|---|---|
| 00:00:00 | A | 00:00:00 | 0 |
| 00:01:00 | A | 00:00:00 | 0 |
| 00:02:00 | B | 00:00:00 | 0 |
| 00:03:00 | D | 00:00:00 | 0 |
| 00:32:00 | B | 00:32:00 | 1 |
| 00:36:00 | C | 00:32:00 | 1 |
| 00:38:00 | D | 00:32:00 | 1 |
| 00:41:00 | E | 00:32:00 | 1 |
| 01:15:00 | A | 01:15:00 | 2 |

## Events between Start and Stop

Find all sequences of events between the event `Start` and the event `Stop` that occur within 5 minutes. Assign a match ID for each sequence.

Run the query

Kusto

```
let Events = datatable (Ts: timespan, Event: string) [
    0m, "A",
    1m, "Start",
    2m, "B",
    3m, "D",
    4m, "Stop",
    6m, "C",
    8m, "Start",
    11m, "E",
    12m, "Stop"
]
;
Events
| sort by Ts asc
| scan with_match_id=m_id with
(
```

```
    step s1: Event == "Start";
    step s2: Event != "Start" and Event != "Stop" and Ts - s1.Ts <= 5m;
    step s3: Event == "Stop" and Ts - s1.Ts <= 5m;
)
```

**Output**

| Ts | Event | m_id |
|---|---|---|
| 00:01:00 | Start | 0 |
| 00:02:00 | B | 0 |
| 00:03:00 | D | 0 |
| 00:04:00 | Stop | 0 |
| 00:08:00 | Start | 1 |
| 00:11:00 | E | 1 |
| 00:12:00 | Stop | 1 |

## Calculate a custom funnel of events

Calculate a funnel completion of the sequence `Hail` -> `Tornado` -> `Thunderstorm Wind` by `State` with custom thresholds on the times between the events (`Tornado` within `1h` and `Thunderstorm Wind` within `2h`). This example is similar to the funnel_sequence_completion plugin, but allows greater flexibility.

Run the query

Kusto

```
StormEvents
| partition hint.strategy=native by State
    (
    sort by StartTime asc
    | scan with
    (
        step hail: EventType == "Hail";
        step tornado: EventType == "Tornado" and StartTime - hail.StartTime
<= 1h;
        step thunderstormWind: EventType == "Thunderstorm Wind" and
StartTime - tornado.StartTime <= 2h;
    )
    )
| summarize dcount(State) by EventType
```

**Output**

| EventType | dcount_State |
|---|---|
| Hail | 50 |
| Tornado | 34 |
| Thunderstorm Wind | 32 |

# Scan logic

`scan` goes over the serialized input data, record by record, comparing each record against each step's condition while taking into account the current state of each step.

## Scan's state

The state that is used behind the scenes by `scan` is a set of records, with the same schema of the output, including source and declared columns. Each step has its own state, the state of step *k* has *k* records in it, where each record in the step's state corresponds to a step up to *k*.

For example, if a scan operator has *n* steps named *s_1*, *s_2*, ..., *s_n* then step *s_k* would have *k* records in its state corresponding to *s_1*, *s_2*, ..., *s_k*. Referencing a value in the state is done in the form *StepName.ColumnName*. For example, `s_2.col1` references column `col1` that belongs to step *s_2* in the state of *s_k*.

## Matching logic

Each record from the input is evaluated against all of scan's steps, starting from last to first. When a record *r* is considered against some step *s_k*, the following logic is applied:

- If the state of the previous step isn't empty and the record *r* satisfies the condition of *s_k* using the state of the previous step *s_(k-1)*, then the following happens:

    1. The state of *s_k* is deleted.
    2. The state of *s_(k-1)* becomes ("promoted" to be) the state of *s_k*, and the state of *s_(k-1)* becomes empty.
    3. All the assignments of *s_k* are calculated and extend *r*.
    4. The extended *r* is added to the output (if *s_k* is defined as `output=all`) and to the state of *s_k*.

- If *r* doesn't satisfy the condition of *s_k* with the state of *s_(k-1)*, *r* is then checked with the state of *s_k*. If *r* satisfies the condition of *s_k* with the state of *s_k*, the following happens:

    1. The record *r* is extended with the assignments of *s_k*.
    2. If *s_k* is defined as `output=all`, the extended record r is added to the output.
    3. The last record in the state of *s_k* (which represents *s_k* itself in the state) is replaced by the extended record *r*.
    4. Whenever the first step is matched while its state is empty, a new match begins and the match ID is increased by `1`. This only affects the output when `with_match_id` is used.

- If r doesn't satisfy the condition *s_k* with the state *s_k*, evaluate *r* against condition *s_k-1* and repeat the logic above.

---

# Feedback

# search operator

Article • 03/23/2023

Searches a text pattern in multiple tables and columns.

> ⓘ Note
>
> If you know the specific tables and columns you want to search, it's more performant to use the **union** and **where** operators. The `search` operator can be slow when searching across a large number of tables and columns.

## Syntax

[*T* `|`] `search` [`kind=` *CaseSensitivity* ] [`in` `(`*TableSources*`)`] *SearchPredicate*

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | | The tabular data source to be searched over, such as a table name, a union operator, or the results of a tabular query. Cannot appear together with *TableSources*. |
| *CaseSensitivity* | string | | A flag that controls the behavior of all `string` scalar operators, such as `has`, with respect to case sensitivity. Valid values are `default`, `case_insensitive`, `case_sensitive`. The options `default` and `case_insensitive` are synonymous, since the default behavior is case insensitive. |
| *TableSources* | string | | A comma-separated list of "wildcarded" table names to take part in the search. The list has the same syntax as the list of the union operator. Cannot appear together with *TabularSource*. |
| *SearchPredicate* | string | ✓ | A boolean expression to be evaluated for every record in the input. If it returns `true`, the record is outputted. See Search predicate syntax. |

### Search predicate syntax

The *SearchPredicate* allows you to search for specific terms in all columns of a table. The operator that will be applied to a search term depends on the presence and placement of a wildcard asterisk (`*`) in the term, as shown in the following table.

| Literal | Operator |
|---|---|
| `billg` | `has` |
| `*billg` | `hassuffix` |
| `billg*` | `hasprefix` |
| `*billg*` | `contains` |
| `bi*lg` | `matches regex` |

You can also restrict the search to a specific column, look for an exact match instead of a term match, or search by regular expression. The syntax for each of these cases is shown in the following table.

| Syntax | Explanation |
|---|---|
| *ColumnName* `:` *StringLiteral* | This syntax can be used to restrict the search to a specific column. The default behavior is to search all columns. |
| *ColumnName* `==` *StringLiteral* | This syntax can be used to search for exact matches of a column against a string value. The default behavior is to look for a term-match. |
| *Column* `matches regex` *StringLiteral* | This syntax indicates regular expression matching, in which *StringLiteral* is the regex pattern. |

Use boolean expressions to combine conditions and create more complex searches. For example, `"error" and x==123` would result in a search for records that have the term `error` in any columns and the value `123` in the `x` column.

> ⓘ Note
>
> If both *TabularSource* and *TableSources* are omitted, the search is carried over all unrestricted tables and views of the database in scope.

## Search predicate syntax examples

| # | Syntax | Meaning (equivalent `where`) | Comments |
|---|---|---|---|

| # | Syntax | Meaning (equivalent `where`) | Comments |
|---|---|---|---|
| 1 | `search "err"` | `where * has "err"` | |
| 2 | `search in (T1,T2,A*) "err"` | `union T1,T2,A* \| where * has "err"` | |
| 3 | `search col:"err"` | `where col has "err"` | |
| 4 | `search col=="err"` | `where col=="err"` | |
| 5 | `search "err*"` | `where * hasprefix "err"` | |
| 6 | `search "*err"` | `where * hassuffix "err"` | |
| 7 | `search "*err*"` | `where * contains "err"` | |
| 8 | `search "Lab*PC"` | `where * matches regex @"\bLab.*PC\b"` | |
| 9 | `search *` | `where 0==0` | |
| 10 | `search col matches regex "..."` | `where col matches regex "..."` | |
| 11 | `search kind=case_sensitive` | | All string comparisons are case-sensitive |
| 12 | `search "abc" and ("def" or "hij")` | `where * has "abc" and (* has "def" or * has hij")` | |
| 13 | `search "err" or (A>a and A<b)` | `where * has "err" or (A>a and A<b)` | |

# Remarks

Unlike the find operator, the `search` operator does not support the following:

1. `withsource=`: The output will always include a column called `$table` of type `string` whose value is the table name from which each record was retrieved (or some system-generated name if the source isn't a table but a composite expression).
2. `project=`, `project-smart`: The output schema is equivalent to `project-smart` output schema.

# Examples

## Global term search

Search for a term over all unrestricted tables and views of the database in scope.

Run the query

```
Kusto
search "Green"
```

The output contains records from the `Customers`, `Products`, and `SalesTable` tables. The `Customers` records shows all customers with the last name "Green", and the `Products` and `SalesTable` records shows products with some mention of "Green".

## Conditional global term search

Search for records that match both terms over all unrestricted tables and views of the database in scope.

Run the query

```
Kusto
search "Green" and ("Deluxe" or "Proseware")
```

## Search a specific table

Search only in the `Customers` table.

Run the query

```
Kusto
search in (Products) "Green"
```

## Case-sensitive search

Search for records that match both case-sensitive terms over all unrestricted tables and views of the database in scope.

Run the query

```
Kusto
search kind=case_sensitive "blue"
```

## Search specific columns

Search for a term in the "FirstName" and "LastName" columns over all unrestricted tables and views of the database in scope.

Run the query

```
Kusto
search FirstName:"Aaron" or LastName:"Hughes"
```

## Limit search by timestamp

Search for a term over all unrestricted tables and views of the database in scope if the term appears in a record with a date greater than the given date.

Run the query

```
Kusto
search "Hughes" and DateKey > datetime('2009-01-01')
```

# Performance Tips

| # | Tip | Prefer | Over |
|---|---|---|---|
| 1 | Prefer to use a single `search` operator over several consecutive `search` operators | `search "billg" and ("steveb" or "satyan")` | `search "billg" \| search "steveb" or "satyan"` |
| 2 | Prefer to filter inside the `search` operator | `search "billg" and "steveb"` | `search * \| where * has "billg" and * has "steveb"` |

# Feedback

**Was this page helpful?** Yes No

Provide product feedback | Get help at Microsoft Q&A

# serialize operator

Article • 03/23/2023

Marks that the order of the input row set is safe to use for window functions.

The operator has a declarative meaning. It marks the input row set as serialized (ordered), so that window functions can be applied to it.

## Syntax

`serialize` [*Name1* `=` *Expr1* [`,` *Name2* `=` *Expr2*]...]

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *Name* | string | | The name of the column to add or update. If omitted, the output column name will be automatically generated. |
| *Expr* | string | ✓ | The calculation to perform over the input. |

## Examples

### Serialize subset of rows by condition

Run the query

Kusto

```
TraceLogs
| where ClientRequestId == "5a848f70-9996-eb17-15ed-21b8eb94bf0e"
| serialize
```

### Add row number to the serialized table

To add a row number to the serialized table, use the row_number() function.

Run the query

Kusto

```
TraceLogs
| where ClientRequestId == "5a848f70-9996-eb17-15ed-21b8eb94bf0e"
| serialize rn = row_number()
```

## Serialization behavior of operators

The output row set of the following operators is marked as serialized.

- getschema
- range
- sort
- top
- top-hitters

The output row set of the following operators is marked as non-serialized.

- count
- distinct
- evaluate
- facet
- join
- make-series
- mv-expand
- reduce by
- sample
- sample-distinct
- summarize
- top-nested

All other operators preserve the serialization property. If the input row set is serialized, then the output row set is also serialized.

---

## Feedback

# shuffle query

Article • 05/01/2023

The `shuffle` query is a semantic-preserving transformation used with a set of operators that support the `shuffle` strategy. Depending on the data involved, querying with the `shuffle` strategy can yield better performance. It is better to use the shuffle query strategy when the `shuffle` key (a `join` key, `summarize` key, `make-series` key or `partition` key) has a high cardinality and the regular operator query hits query limits.

You can use the following operators with the shuffle command:

- join
- summarize
- make-series
- partition

To use the `shuffle` query strategy, add the expression `hint.strategy = shuffle` or `hint.shufflekey = <key>`. When you use `hint.strategy=shuffle`, the operator data will be shuffled by all the keys. Use this expression when the compound key is unique but each key is not unique enough, so you will shuffle the data using all the keys of the shuffled operator.

When partitioning data with the shuffle strategy, the data load is shared on all cluster nodes. Each node processes one partition of the data. The default number of partitions is equal to the number of cluster nodes.

The partition number can be overridden by using the syntax `hint.num_partitions = total_partitions`, which will control the number of partitions. This is useful when the cluster has a small number of cluster nodes and the default partitions number will be small, and the query fails or takes a long execution time.

> ⓘ Note
>
> Using many partitions may consume more cluster resources and degrade performance. Choose the partition number carefully by starting with the `hint.strategy = shuffle` and start increasing the partitions gradually.

In some cases, the `hint.strategy = shuffle` will be ignored, and the query will not run in `shuffle` strategy. This can happen when:

- The `join` operator has another `shuffle`-compatible operator (`join`, `summarize`, `make-series` or `partition`) on the left side or the right side.
- The `summarize` operator appears after another `shuffle`-compatible operator (`join`, `summarize`, `make-series` or `partition`) in the query.

# Syntax

## With `hint.strategy` = `shuffle`

*T* `|` *DataExpression* `|` `join` `hint.strategy` = `shuffle` `(` *DataExpression* `)`

*T* `|` `summarize` `hint.strategy` = `shuffle` *DataExpression*

*T* `|` *Query* `|` partition `hint.strategy` = `shuffle` `(` *SubQuery* `)`

## With `hint.shufflekey` = *key*

*T* `|` *DataExpression* `|` `join` `hint.shufflekey` = *key* `(` *DataExpression* `)`

*T* `|` `summarize` `hint.shufflekey` = *key DataExpression*

*T* `|` `make-series` `hint.shufflekey` = *key DataExpression*

*T* `|` *Query* `|` partition `hint.shufflekey` = *key* `(` *SubQuery* `)`

# Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | ✓ | The tabular source whose data is to be processed by the operator. |
| *DataExpression* | string | | An implicit or explicit tabular transformation expression. |
| *Query* | string | | A transformation expression run on the records of *T*. |
| *key* | string | | Use a `join` key, `summarize` key, `make-series` key or `partition` key. |
| *SubQuery* | string | | A transformation expression. |

> ⓘ Note

Either *DataExpression* or *Query* must be specified depending on the chosen syntax.

# Examples

## Use summarize with shuffle

The `shuffle` strategy query with `summarize` operator will share the load on all cluster nodes, where each node will process one partition of the data.

Run the query

Kusto

```
StormEvents
| summarize hint.strategy = shuffle count(), avg(InjuriesIndirect) by State
| count
```

**Output**

| Count |
| --- |
| 67 |

## Use join with shuffle

Run the query

Kusto

```
StormEvents
| where State has "West"
| where EventType has "Flood"
| join hint.strategy=shuffle
    (
    StormEvents
    | where EventType has "Hail"
    | project EpisodeId, State, DamageProperty
    )
    on State
| count
```

**Output**

| Count |
| --- |
| 103 |

# Use make-series with shuffle

Run the query

Kusto

```
StormEvents
| where State has "North"
| make-series hint.shufflekey = State sum(DamageProperty) default = 0 on
StartTime in range(datetime(2007-01-01 00:00:00.0000000), datetime(2007-01-
31 23:59:00.0000000), 15d) by State
```

### Output

| State | sum_DamageProperty | StartTime |
| --- | --- | --- |
| NORTH DAKOTA | [60000,0,0] | ["2006-12-31T00:00:00.0000000Z","2007-01-15T00:00:00.0000000Z","2007-01-30T00:00:00.0000000Z"] |
| NORTH CAROLINA | [20000,0,1000] | ["2006-12-31T00:00:00.0000000Z","2007-01-15T00:00:00.0000000Z","2007-01-30T00:00:00.0000000Z"] |
| ATLANTIC NORTH | [0,0,0] | ["2006-12-31T00:00:00.0000000Z","2007-01-15T00:00:00.0000000Z","2007-01-30T00:00:00.0000000Z"] |

# Use partition with shuffle

Run the query

Kusto

```
StormEvents
| partition hint.strategy=shuffle by EpisodeId
(
    top 3 by DamageProperty
    | project EpisodeId, State, DamageProperty
)
| count
```

### Output

| Count |
| --- |
| 22345 |

## Compare hint.strategy=shuffle and hint.shufflekey=key

When you use `hint.strategy=shuffle`, the shuffled operator will be shuffled by all the keys. In the following example, the query shuffles the data using both `EpisodeId` and `EventId` as keys:

Run the query

Kusto

```
StormEvents
| where StartTime > datetime(2007-01-01 00:00:00.0000000)
| join kind = inner hint.strategy=shuffle (StormEvents | where DamageCrops >
62000000) on EpisodeId, EventId
| count
```

**Output**

| Count |
| --- |
| 14 |

The following query uses `hint.shufflekey = key`. The query above is equivalent to this query.

Run the query

Kusto

```
StormEvents
| where StartTime > datetime(2007-01-01 00:00:00.0000000)
| join kind = inner hint.shufflekey = EpisodeId hint.shufflekey = EventId
(StormEvents | where DamageCrops > 62000000) on EpisodeId, EventId
```

**Output**

| Count |
| --- |
| 14 |

## Shuffle the data with multiple keys

In some cases, the `hint.strategy=shuffle` will be ignored, and the query will not run in shuffle strategy. For example, in the following example, the join has summarize on its left side, so using `hint.strategy=shuffle` will not apply shuffle strategy to the query:

Run the query

Kusto

```
StormEvents
| where StartTime > datetime(2007-01-01 00:00:00.0000000)
| summarize count() by EpisodeId, EventId
| join kind = inner hint.strategy=shuffle (StormEvents | where DamageCrops >
62000000) on EpisodeId, EventId
```

### Output

| EpisodeId | EventId | ... | EpisodeId1 | EventId1 | ... |
|---|---|---|---|---|---|
| 1030 | 4407 | ... | 1030 | 4407 | ... |
| 1030 | 13721 | ... | 1030 | 13721 | ... |
| 2477 | 12530 | ... | 2477 | 12530 | ... |
| 2103 | 10237 | ... | 2103 | 10237 | ... |
| 2103 | 10239 | ... | 2103 | 10239 | ... |
| ... | ... | ... | ... | ... | ... |

To overcome this issue and run in shuffle strategy, choose the key which is common for the `summarize` and `join` operations. In this case, this key is `EpisodeId`. Use the hint `hint.shufflekey` to specify the shuffle key on the `join` to `hint.shufflekey = EpisodeId`:

Run the query

Kusto

```
StormEvents
| where StartTime > datetime(2007-01-01 00:00:00.0000000)
| summarize count() by EpisodeId, EventId
| join kind = inner hint.shufflekey=EpisodeId (StormEvents | where
DamageCrops > 62000000) on EpisodeId, EventId
```

**Output**

| EpisodeId | EventId | ... | EpisodeId1 | EventId1 | ... |
|---|---|---|---|---|---|
| 1030 | 4407 | ... | 1030 | 4407 | ... |
| 1030 | 13721 | ... | 1030 | 13721 | ... |
| 2477 | 12530 | ... | 2477 | 12530 | ... |
| 2103 | 10237 | ... | 2103 | 10237 | ... |
| 2103 | 10239 | ... | 2103 | 10239 | ... |
| ... | ... | ... | ... | ... | ... |

## Use summarize with shuffle to improve performance

In this example, using the `summarize` operator with `shuffle` strategy improves performance. The source table has 150M records and the cardinality of the group by key is 10M, which is spread over 10 cluster nodes.

Using `summarize` operator without `shuffle` strategy, the query ends after 1:08 and the memory usage peak is ~3 GB:

Kusto

```
orders
| summarize arg_max(o_orderdate, o_totalprice) by o_custkey
| where o_totalprice < 1000
| count
```

**Output**

| Count |
|---|
| 1086 |

While using `shuffle` strategy with `summarize`, the query ends after ~7 seconds and the memory usage peak is 0.43 GB:

Kusto

```
orders
| summarize hint.strategy = shuffle arg_max(o_orderdate, o_totalprice) by
o_custkey
```

```
| where o_totalprice < 1000
| count
```

**Output**

| Count |
| --- |
| 1086 |

The following example demonstrates performance on a cluster that has two cluster nodes, with a table that has 60M records, where the cardinality of the group by key is 2M.

Running the query without `hint.num_partitions` will use only two partitions (as cluster nodes number) and the following query will take ~1:10 mins:

Kusto

```
lineitem
| summarize hint.strategy = shuffle dcount(l_comment), dcount(l_shipdate) by
l_partkey
| consume
```

If setting the partitions number to 10, the query will end after 23 seconds:

Kusto

```
lineitem
| summarize hint.strategy = shuffle hint.num_partitions = 10
dcount(l_comment), dcount(l_shipdate) by l_partkey
| consume
```

# Use join with shuffle to improve performance

The following example shows how using `shuffle` strategy with the `join` operator improves performance.

The examples were sampled on a cluster with 10 nodes where the data is spread over all these nodes.

The query's left-side source table has 15M records where the cardinality of the `join` key is ~14M. The query's right-side source has 150M records and the cardinality of the `join` key is 10M. The query ends after ~28 seconds and the memory usage peak is 1.43 GB:

Kusto

```
customer
| join
    orders
on $left.c_custkey == $right.o_custkey
| summarize sum(c_acctbal) by c_nationkey
```

When using `shuffle` strategy with a `join` operator, the query ends after ~4 seconds and the memory usage peak is 0.3 GB:

Kusto

```
customer
| join
    hint.strategy = shuffle orders
on $left.c_custkey == $right.o_custkey
| summarize sum(c_acctbal) by c_nationkey
```

In another example, we try the same queries on a larger dataset with the following conditions:

- Left-side source of the `join` is 150M and the cardinality of the key is 148M.
- Right-side source of the `join` is 1.5B, and the cardinality of the key is ~100M.

The query with just the `join` operator hits limits and times-out after 4 mins. However, when using `shuffle` strategy with the `join` operator, the query ends after ~34 seconds and the memory usage peak is 1.23 GB.

The following example shows the improvement on a cluster that has two cluster nodes, with a table of 60M records, where the cardinality of the `join` key is 2M. Running the query without `hint.num_partitions` will use only two partitions (as cluster nodes number) and the following query will take ~1:10 mins:

Kusto

```
lineitem
| summarize dcount(l_comment), dcount(l_shipdate) by l_partkey
| join
    hint.shufflekey = l_partkey   part
on $left.l_partkey == $right.p_partkey
| consume
```

When setting the partitions number to 10, the query will end after 23 seconds:

Kusto

```
lineitem
| summarize dcount(l_comment), dcount(l_shipdate) by l_partkey
| join
    hint.shufflekey = l_partkey  hint.num_partitions = 10    part
on $left.l_partkey == $right.p_partkey
| consume
```

## Feedback

**Was this page helpful?** Yes No

Provide product feedback | Get help at Microsoft Q&A

# summarize operator

Article • 03/23/2023

Produces a table that aggregates the content of the input table.

## Syntax

*T* `| summarize` [ *SummarizeParameters* ] [[*Column* `=`] *Aggregation* [`,` ...]] [`by` [*Column* `=`] *GroupExpression* [`,` ...]]

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *Column* | string | | The name for the result column. Defaults to a name derived from the expression. |
| *Aggregation* | string | ✓ | A call to an aggregation function such as `count()` or `avg()`, with column names as arguments. |
| *GroupExpression* | scalar | ✓ | A scalar expression that can reference the input data. The output will have as many records as there are distinct values of all the group expressions. |
| *SummarizeParameters* | string | | Zero or more space-separated parameters in the form of *Name* `=` *Value* that control the behavior. See supported parameters. |

> ⓘ Note
>
> When the input table is empty, the output depends on whether *GroupExpression* is used:
>
> - If *GroupExpression* is not provided, the output will be a single (empty) row.
> - If *GroupExpression* is provided, the output will have no rows.

### Supported parameters

| Name | Description |
|---|---|
| `hint.num_partitions` | Specifies the number of partitions used to share the query load on cluster nodes. See shuffle query |
| `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 will process one partition of the data. See shuffle query |

## Returns

The input rows are arranged into groups having the same values of the `by` expressions. Then the specified aggregation functions are computed over each group, producing a row for each group. The result contains the `by` columns and also at least one column for each computed aggregate. (Some aggregation functions return multiple columns.)

The result has as many rows as there are distinct combinations of `by` values (which may be zero). If there are no group keys provided, the result has a single record.

To summarize over ranges of numeric values, use `bin()` to reduce ranges to discrete values.

> ⓘ Note
>
> - Although you can provide arbitrary expressions for both the aggregation and grouping expressions, it's more efficient to use simple column names, or apply `bin()` to a numeric column.
> - The automatic hourly bins for datetime columns is no longer supported. Use explicit binning instead. For example, `summarize by bin(timestamp, 1h)`.

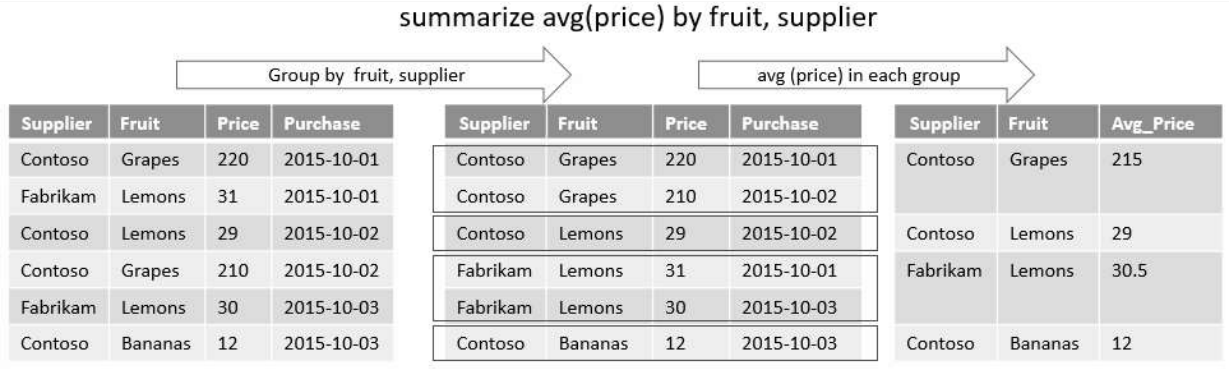
# Aggregates default values

The following table summarizes the default values of aggregations:

| Operator | Default value |
|---|---|
| `count()`, `countif()`, `dcount()`, `dcountif()` | 0 |
| `make_bag()`, `make_bag_if()`, `make_list()`, `make_list_if()`, `make_set()`, `make_set_if()` | empty dynamic array ([]) |
| All others | null |

When using these aggregates over entities that includes null values, the null values will be ignored and won't participate in the calculation (see examples below).

# Examples




| Supplier | Fruit | Price | Purchase |
|---|---|---|---|
| Contoso | Grapes | 220 | 2015-10-01 |
| Fabrikam | Lemons | 31 | 2015-10-01 |
| Contoso | Lemons | 29 | 2015-10-02 |
| Contoso | Grapes | 210 | 2015-10-02 |
| Fabrikam | Lemons | 30 | 2015-10-03 |
| Contoso | Bananas | 12 | 2015-10-03 |

| Supplier | Fruit | Price | Purchase |
|---|---|---|---|
| Contoso | Grapes | 220 | 2015-10-01 |
| Contoso | Grapes | 210 | 2015-10-02 |
| Contoso | Lemons | 29 | 2015-10-02 |
| Fabrikam | Lemons | 31 | 2015-10-01 |
| Fabrikam | Lemons | 30 | 2015-10-03 |
| Contoso | Bananas | 12 | 2015-10-03 |

| Supplier | Fruit | Avg_Price |
|---|---|---|
| Contoso | Grapes | 215 |
| Contoso | Lemons | 29 |
| Fabrikam | Lemons | 30.5 |
| Contoso | Bananas | 12 |

## Unique combination

The following query determines what unique combinations of `State` and `EventType` there are for storms that resulted in direct injury. There are no aggregation functions, just group-by keys. The output will just show the columns for those results.

Run the query

Kusto

```
StormEvents
| where InjuriesDirect > 0
| summarize by State, EventType
```

**Output**

The following table shows only the first 5 rows. To see the full output, run the query.

| State | EventType |
|---|---|
| TEXAS | Thunderstorm Wind |
| TEXAS | Flash Flood |
| TEXAS | Winter Weather |
| TEXAS | High Wind |
| TEXAS | Flood |
| ... | ... |

## Minimum and maximum timestamp

Finds the minimum and maximum heavy rain storms in Hawaii. There's no group-by clause, so there's just one row in the output.

Run the query

```
Kusto

StormEvents
| where State == "HAWAII" and EventType == "Heavy Rain"
| project Duration = EndTime - StartTime
| summarize Min = min(Duration), Max = max(Duration)
```

**Output**

| Min | Max |
|---|---|
| 01:08:00 | 11:55:00 |

## Distinct count

Create a row for each continent, showing a count of the cities in which activities occur. Because there are few values for "continent", no grouping function is needed in the 'by' clause:

Run the query

```
Kusto

StormEvents
| summarize TypesOfStorms=dcount(EventType) by State
| sort by TypesOfStorms
```

**Output**

The following table shows only the first 5 rows. To see the full output, run the query.

| State | TypesOfStorms |
|---|---|
| TEXAS | 27 |
| CALIFORNIA | 26 |
| PENNSYLVANIA | 25 |
| GEORGIA | 24 |
| ILLINOIS | 23 |
| ... | ... |

## Histogram

The following example calculates a histogram storm event types that had storms lasting longer than 1 day. Because `Duration` has many values, use `bin()` to group its values into 1-day intervals.

Run the query

```
Kusto

StormEvents
| project EventType, Duration = EndTime - StartTime
| where Duration > 1d
| summarize EventCount=count() by EventType, Length=bin(Duration, 1d)
| sort by Length
```

**Output**

| EventType | Length | EventCount |
|---|---|---|

| EventType | Length | EventCount |
|---|---|---|
| Drought | 30.00:00:00 | 1646 |
| Wildfire | 30.00:00:00 | 11 |
| Heat | 30.00:00:00 | 14 |
| Flood | 30.00:00:00 | 20 |
| Heavy Rain | 29.00:00:00 | 42 |
| ... | ... | ... |

## Aggregates default values

When the input of `summarize` operator has at least one empty group-by key, its result is empty, too.

When the input of `summarize` operator doesn't have an empty group-by key, the result is the default values of the aggregates used in the `summarize`:

Run the query

Kusto

```
datatable(x:long)[]
| summarize any_x=take_any(x), arg_max_x=arg_max(x, *), arg_min_x=arg_min(x, *), avg(x),
buildschema(todynamic(tostring(x))), max(x), min(x), percentile(x, 55), hll(x) ,stdev(x), sum(x), sumif(x, x > 0),
tdigest(x), variance(x)
```

**Output**

| any_x | arg_max_x | arg_min_x | avg_x | schema_x | max_x | min_x | percentile_x_55 | hll_x | stdev_x | sum_x | sumif_x | tdigest_x | variance_x |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | NaN | | | | | | 0 | 0 | 0 | | 0 |

The result of `avg_x(x)` is `NaN` due to dividing by 0.

Run the query

Kusto

```
datatable(x:long)[]
| summarize  count(x), countif(x > 0) , dcount(x), dcountif(x, x > 0)
```

**Output**

| count_x | countif_ | dcount_x | dcountif_x |
|---|---|---|---|
| 0 | 0 | 0 | 0 |

Run the query

Kusto

```
datatable(x:long)[]
| summarize  make_set(x), make_list(x)
```

**Output**

| set_x | list_x |
|---|---|
| [] | [] |

The aggregate avg sums all the non-nulls and counts only those which participated in the calculation (won't take nulls into account).

Run the query

```
Kusto
range x from 1 to 2 step 1
| extend y = iff(x == 1, real(null), real(5))
| summarize sum(y), avg(y)
```

**Output**

| sum_y | avg_y |
|---|---|
| 5 | 5 |

The regular count will count nulls:

Run the query

```
Kusto
range x from 1 to 2 step 1
| extend y = iff(x == 1, real(null), real(5))
| summarize count(y)
```

**Output**

| count_y |
|---|
| 2 |

Run the query

```
Kusto
range x from 1 to 2 step 1
| extend y = iff(x == 1, real(null), real(5))
| summarize make_set(y), make_set(y)
```

**Output**

| set_y | set_y1 |
|---|---|
| [5.0] | [5.0] |

# Feedback

# Using hll() and tdigest()

Article • 03/02/2023

Suppose you want to calculate the count of distinct users every day over the last seven days. You can run `summarize dcount(user)` once a day with a span filtered to the last seven days. This method is inefficient, because each time the calculation is run, there's a six-day overlap with the previous calculation. You can also calculate an aggregate for each day, and then combine these aggregates. This method requires you to "remember" the last six results, but it's much more efficient.

Partitioning queries as described is easy for simple aggregates, such as `count()` and `sum()`. It can also be useful for complex aggregates, such as `dcount()` and `percentiles()`. This article explains how Kusto supports such calculations.

The following examples show how to use `hll`/`tdigest` and demonstrate that using these commands is highly performant in some scenarios:

> ⓘ **Important**
>
> The precise binary representation of hll(), hll_if(), hll_merge(), tdigest(), and tdigest_merge() results may change over time. There's no guarantee that these functions will produce identical results for identical inputs, and therefore we don't advise relying on them.

> ⓘ **Note**
>
> In some cases, the dynamic objects generated by the `hll` or the `tdigest` aggregate functions may be big and exceed the default MaxValueSize property in the encoding policy. If so, the object will be ingested as null. For example, when persisting the output of `hll` function with accuracy level 4, the size of the `hll` object exceeds the default MaxValueSize, which is 1MB. To avoid this issue, modify the encoding policy of the column as shown in the following examples.

Run the query

Kusto

```
range x from 1 to 1000000 step 1
| summarize hll(x,4)
| project sizeInMb = estimate_data_size(hll_x) / pow(1024,2)
```

**Output**

| sizeInMb |
| --- |
| 1.0000524520874 |

Ingesting this object into a table before applying this kind of policy will ingest null:

```
Kusto

.set-or-append MyTable <| range x from 1 to 1000000 step 1
| summarize hll(x,4)
```

```
Kusto

MyTable
| project isempty(hll_x)
```

**Output**

| Column1 |
| --- |
| 1 |

To avoid ingesting null, use the special encoding policy type `bigobject`, which overrides the `MaxValueSize` to 2 MB like this:

```
Kusto

.alter column MyTable.hll_x policy encoding type='bigobject'
```

Ingesting a value now to the same table above:

```
Kusto

.set-or-append MyTable <| range x from 1 to 1000000 step 1
| summarize hll(x,4)
```

ingests the second value successfully:

```
Kusto

MyTable
| project isempty(hll_x)
```

**Output**

| Column1 |
| --- |
| 1 |
| 0 |

# Example: Count with binned timestamp

There's a table, `PageViewsHllTDigest`, containing `hll` values of Pages viewed in each hour. You want these values binned to `12h`. Merge the `hll` values using the `hll_merge()` aggregate function, with the timestamp binned to `12h`. Use the function `dcount_hll` to return the final `dcount` value:

Kusto

```
PageViewsHllTDigest
| summarize merged_hll = hll_merge(hllPage) by bin(Timestamp, 12h)
| project Timestamp , dcount_hll(merged_hll)
```

**Output**

| Timestamp | `dcount_hll_merged_hll` |
| --- | --- |
| 2016-05-01 12:00:00.0000000 | 20056275 |
| 2016-05-02 00:00:00.0000000 | 38797623 |
| 2016-05-02 12:00:00.0000000 | 39316056 |
| 2016-05-03 00:00:00.0000000 | 13685621 |

To bin timestamp for `1d`:

Kusto

```
PageViewsHllTDigest
| summarize merged_hll = hll_merge(hllPage) by bin(Timestamp, 1d)
| project Timestamp , dcount_hll(merged_hll)
```

**Output**

| Timestamp | `dcount_hll_merged_hll` |
|---|---|
| 2016-05-01 00:00:00.0000000 | 20056275 |
| 2016-05-02 00:00:00.0000000 | 64135183 |
| 2016-05-03 00:00:00.0000000 | 13685621 |

The same query may be done over the values of `tdigest`, which represent the `BytesDelivered` in each hour:

Kusto

```
PageViewsHllTDigest
| summarize merged_tdigests = merge_tdigest(tdigestBytesDel) by
bin(Timestamp, 12h)
| project Timestamp , percentile_tdigest(merged_tdigests, 95, typeof(long))
```

**Output**

| Timestamp | `percentile_tdigest_merged_tdigests` |
|---|---|
| 2016-05-01 12:00:00.0000000 | 170200 |
| 2016-05-02 00:00:00.0000000 | 152975 |
| 2016-05-02 12:00:00.0000000 | 181315 |
| 2016-05-03 00:00:00.0000000 | 146817 |

# Example: Temporary table

Kusto limits are reached with datasets that are too large, where you need to run periodic queries over the dataset, but run the regular queries to calculate percentile() or dcount() over large datasets.

To solve this problem, newly added data may be added to a temp table as `hll` or `tdigest` values using hll() when the required operation is `dcount` or tdigest() when the required operation is percentile using set/append or update policy. In this case, the intermediate results of `dcount` or `tdigest` are saved into another dataset, which should be smaller than the target large one.

When you need to get the final results of these values, the queries may use `hll`/`tdigest` mergers: hll-merge()/tdigest_merge(). Then, after getting the merged

values, percentile_tdigest() / dcount_hll() may be invoked on these merged values to get the final result of `dcount` or percentiles.

Assuming there's a table, PageViews, into which data is ingested daily, every day on which you want to calculate the distinct count of pages viewed per minute later than date = datetime(2016-05-01 18:00:00.0000000).

Run the following query:

Kusto

```
PageViews
| where Timestamp > datetime(2016-05-01 18:00:00.0000000)
| summarize percentile(BytesDelivered, 90), dcount(Page,2) by bin(Timestamp,
1d)
```

**Output**

| Timestamp | percentile_BytesDelivered_90 | dcount_Page |
|---|---|---|
| 2016-05-01 00:00:00.0000000 | 83634 | 20056275 |
| 2016-05-02 00:00:00.0000000 | 82770 | 64135183 |
| 2016-05-03 00:00:00.0000000 | 72920 | 13685621 |

This query aggregates all the values every time you run this query (for example, if you want to run it many times a day).

If you save the `hll` and `tdigest` values (which are the intermediate results of `dcount` and percentile) into a temp table, `PageViewsHllTDigest`, using an update policy or set/append commands, you may only merge the values and then use `dcount_hll`/`percentile_tdigest` using the following query:

Kusto

```
PageViewsHllTDigest
| summarize  percentile_tdigest(merge_tdigest(tdigestBytesDel), 90),
dcount_hll(hll_merge(hllPage)) by bin(Timestamp, 1d)
```

**Output**

| Timestamp | `percentile_tdigest_merge_tdigests_tdigestBytesDel` | `dcount_hll_hll_merge_hllPage` |
|---|---|---|
| 2016-05-01 00:00:00.0000000 | 84224 | 20056275 |

| Timestamp | `percentile_tdigest_merge_tdigests_tdigestBytesDel` | `dcount_hll_hll_merge_hllPage` |
|---|---|---|
| 2016-05-02 00:00:00.0000000 | 83486 | 64135183 |
| 2016-05-03 00:00:00.0000000 | 72247 | 13685621 |

This query should be more performant, as it runs over a smaller table. In this example, the first query runs over ~215M records, while the second one runs over just 32 records:

# Example: Intermediate results

The Retention Query. Assume you have a table that summarizes when each Wikipedia page was viewed (sample size is 10M), and you want to find for each date1 date2 the percentage of pages reviewed in both date1 and date2 relative to the pages viewed on date1 (date1 < date2).

The trivial way uses join and summarize operators:

Kusto

```
// Get the total pages viewed each day
let totalPagesPerDay = PageViewsSample
| summarize by Page, Day = startofday(Timestamp)
| summarize count() by Day;
// Join the table to itself to get a grid where
// each row shows foreach page1, in which two dates
// it was viewed.
// Then count the pages between each two dates to
// get how many pages were viewed between date1 and date2.
PageViewsSample
| summarize by Page, Day1 = startofday(Timestamp)
| join kind = inner
(
    PageViewsSample
    | summarize by Page, Day2 = startofday(Timestamp)
)
on Page
| where Day2 > Day1
| summarize count() by Day1, Day2
| join kind = inner
    totalPagesPerDay
on $left.Day1 == $right.Day
| project Day1, Day2, Percentage = count_*100.0/count_1
```

**Output**

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

The above query took ~18 seconds.

When you use the hll(), hll_merge(), and dcount_hll() functions, the equivalent query will end after ~1.3 seconds and show that the `hll` functions speeds up the query above by ~14 times:

Kusto

```
let Stats=PageViewsSample | summarize pagehll=hll(Page, 2) by
day=startofday(Timestamp); // saving the hll values (intermediate results of
the dcount values)
let day0=toscalar(Stats | summarize min(day)); // finding the min date over
all dates.
let dayn=toscalar(Stats | summarize max(day)); // finding the max date over
all dates.
let daycount=tolong((dayn-day0)/1d); // finding the range between max and
min
Stats
| project idx=tolong((day-day0)/1d), day, pagehll
| mv-expand pidx=range(0, daycount) to typeof(long)
// Extend the column to get the dcount value from hll'ed values for each
date (same as totalPagesPerDay from the above query)
| extend key1=iff(idx < pidx, idx, pidx), key2=iff(idx < pidx, pidx, idx),
pages=dcount_hll(pagehll)
// For each two dates, merge the hll'ed values to get the total dcount over
each two dates,
// This helps to get the pages viewed in both date1 and date2 (see the
description below about the intersection_size)
| summarize (day1, pages1)=arg_min(day, pages), (day2, pages2)=arg_max(day,
pages), union_size=dcount_hll(hll_merge(pagehll)) by key1, key2
| where day2 > day1
// To get pages viewed in date1 and also date2, look at the merged dcount of
date1 and date2, subtract it from pages of date1 + pages on date2.
| project pages1, day1,day2, intersection_size=(pages1 + pages2 -
union_size)
| project day1, day2, Percentage = intersection_size*100.0 / pages1
```

### Output

| day1 | day2 | Percentage |
|---|---|---|
| 2016-05-01 00:00:00.0000000 | 2016-05-02 00:00:00.0000000 | 33.2298494510578 |

| day1 | day2 | Percentage |
|---|---|---|
| 2016-05-01 00:00:00.0000000 | 2016-05-03 00:00:00.0000000 | 16.9773830213667 |
| 2016-05-02 00:00:00.0000000 | 2016-05-03 00:00:00.0000000 | 14.5160020350006 |

> **Note**
>
> The results of the queries are not 100% accurate due to the error of the `hll` functions. For more information about the errors, see **dcount()**.

# Feedback

# take operator

Article • 05/24/2023

Return up to the specified number of rows.

There is no guarantee which records are returned, unless the source data is sorted. If the data is sorted, then the top values will be returned.

> The `take` and `limit` operators are equivalent

> **Note**
>
> `take` is a simple, quick, and efficient way to view a small sample of records when browsing data interactively, but be aware that it doesn't guarantee any consistency in its results when executing multiple times, even if the data set hasn't changed. Even if the number of rows returned by the query isn't explicitly limited by the query (no `take` operator is used), Kusto limits that number by default. For more details, see **Kusto query limits**.

## Syntax

`take` *NumberOfRows*

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *NumberOfRows* | int | ✓ | The number of rows to return. |

## Paging of query results

Methods for implementing paging include:

- Export the result of a query to an external storage and paging through the generated data.
- Write a middle-tier application that provides a stateful paging API by caching the results of a Kusto query.
- Use pagination in Stored query results.

# Example

Run the query

```
Kusto
StormEvents | take 5
```

# See also

- sort operator
- top operator
- top-nested operator

---

# Feedback

# top operator

Article • 02/27/2023

Returns the first *N* records sorted by the specified columns.

## Syntax

*T* `| top` *NumberOfRows* `by` *Expression* [`asc` | `desc`] [`nulls first` | `nulls last`]

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | ✓ | The tabular input to sort. |
| *NumberOfRows* | int | ✓ | The number of rows of *T* to return. |
| *Expression* | string | ✓ | The scalar expression by which to sort. |
| `asc` or `desc` | string | | Controls whether the selection is from the "bottom" or "top" of the range. Default `desc`. |
| `nulls first` or `nulls last` | string | | Controls whether null values appear at the "bottom" or "top" of the range. Default for `asc` is `nulls first`. Default for `desc` is `nulls last`. |

> **Tip**
>
> `top 5 by name` is equivalent to the expression `sort by name | take 5` both from semantic and performance perspectives.

## Example

Show top three storms with most direct injuries.

Run the query

Kusto

```
StormEvents
| top 3 by InjuriesDirect
```

The below table shows only the relevant column. Run the query above to see more storm details for these events.

| InjuriesDirect | ... |
|---|---|
| 519 | ... |
| 422 | ... |
| 200 | ... |

## See also

- Use top-nested operator to produce hierarchical (nested) top results.

## Feedback

# top-nested operator

Article • 03/30/2023

Produces a hierarchical aggregation and top values selection, where each level is a refinement of the previous one.

Kusto

```
T | top-nested 3 of Location with others="Others" by sum(MachinesNumber), top-nested 4 of bin(Timestamp,5m) by sum(MachinesNumber)
```

The `top-nested` operator accepts tabular data as input, and one or more aggregation clauses. The first aggregation clause (left-most) subdivides the input records into partitions, according to the unique values of some expression over those records. The clause then keeps a certain number of records that maximize or minimize this expression over the records. The next aggregation clause then applies a similar function, in a nested fashion. Each following clause is applied to the partition produced by the previous clause. This process continues for all aggregation clauses.

For example, the `top-nested` operator can be used to answer the following question: "For a table containing sales figures, such as country/region, salesperson, and amount sold: what are the top five countries/regions by sales? What are the top three salespeople in each of these countries/regions?"

## Syntax

*T* `|` `top-nested` *TopNestedClause* [`,` `top-nested` *TopNestedClause2*]...

Where *TopNestedClause* has the following syntax:

[ *N* ] `of` [*ExprName* `=`] *Expr* [`with` `others` `=` *ConstExpr*] `by` [*AggName* `=`] *Aggregation* [`asc` | `desc`]

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | ✓ | The input tabular expression. |
| *N* | long | | The number of top values to return for this hierarchy level. If omitted, all distinct values will be returned. |
| *ExprName* | string | | If specified, sets the name of the output column corresponding to the values of *Expr*. |
| *Expr* | string | ✓ | An expression over the input record indicating which value to return for this hierarchy level. Typically it's a column reference from *T*, or some calculation, such as `bin()`, over such a column. |
| *ConstExpr* | string | | If specified, for each hierarchy level, 1 record will be added with the value that is the aggregation over all records that didn't "make it to the top". |
| *AggName* | string | | If specified, this identifier sets the column name in the output for the value of *Aggregation*. |

| Name | Type | Required | Description |
|---|---|---|---|
| *Aggregation* | string | | The aggregation function to apply to all records sharing the same value of *Expr*. The value of this aggregation determines which of the resulting records are "top". For the possible values, see supported aggregation functions. |
| `asc` or `desc` | string | | Controls whether selection is actually from the "bottom" or "top" of the range of aggregated values. The default is `desc`. |

## Supported aggregation functions

The following aggregation functions are supported:

- sum()
- count()
- max()
- min()
- dcount()
- avg()
- percentile()
- percentilew()

> ⓘ **Note**
>
> Any algebraic combination of the aggregations is also supported.

# Returns

This operator returns a table that has two columns for each aggregation clause:

- One column holds the distinct values of the clause's `Expr` calculation (having the column name *ExprName* if specified)
- One column holds the result of the *Aggregation* calculation (having the column name *AggregationName* if specified)

# Notes

Input columns that aren't specified as `Expr` values aren't outputted. To get all values at a certain level, add an aggregation count that:

- Omits the value of *N*
- Uses the column name as the value of `Expr`
- Uses `Ignore=max(1)` as the aggregation, and then ignore (or project-away) the column `Ignore`.

The number of records may grow exponentially with the number of aggregation clauses ((N1+1) * (N2+1) * ...). Record growth is even faster if no *N* limit is specified. Take into account that this operator may consume a considerable amount of resources.

If the distribution of the aggregation is considerably non-uniform, limit the number of distinct values to return (by using *N*) and use the `with others=` *ConstExpr* option to get an indication for the "weight" of all other cases.

# Examples

Run the query

Kusto

```
StormEvents
| top-nested 2 of State       by sum(BeginLat),
  top-nested 3 of Source      by sum(BeginLat),
  top-nested 1 of EndLocation by sum(BeginLat)
```

**Output**

| State | aggregated_State | Source | aggregated_Source | EndLocation | aggregated_EndLocation |
|---|---|---|---|---|---|
| KANSAS | 87771.2355000001 | Law Enforcement | 18744.823 | FT SCOTT | 264.858 |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | BUCKLIN | 488.2457 |
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | SHARON SPGS | 388.7404 |
| TEXAS | 123400.5101 | Public | 13650.9079 | AMARILLO | 246.2598 |
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | PERRYTON | 289.3178 |
| TEXAS | 123400.5101 | Trained Spotter | 13997.7124 | CLAUDE | 421.44 |

Use the option 'with others':

Run the query

Kusto

```
StormEvents
| top-nested 2 of State with others = "All Other States" by sum(BeginLat),
  top-nested 3 of Source by sum(BeginLat),
  top-nested 1 of EndLocation with others = "All Other End Locations" by sum(BeginLat)
```

**Output**

| State | aggregated_State | Source | aggregated_Source | EndLocation | aggregated_EndLocation |
|---|---|---|---|---|---|
| KANSAS | 87771.2355000001 | Law Enforcement | 18744.823 | FT SCOTT | 264.858 |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | BUCKLIN | 488.2457 |
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | SHARON SPGS | 388.7404 |
| TEXAS | 123400.5101 | Public | 13650.9079 | AMARILLO | 246.2598 |

| State | aggregated_State | Source | aggregated_Source | EndLocation | aggregated_EndLocation |
|---|---|---|---|---|---|
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | PERRYTON | 289.3178 |
| TEXAS | 123400.5101 | Trained Spotter | 13997.7124 | CLAUDE | 421.44 |
| KANSAS | 87771.2355000001 | Law Enforcement | 18744.823 | All Other End Locations | 18479.965 |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | All Other End Locations | 22367.3749 |
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | All Other End Locations | 20890.9679 |
| TEXAS | 123400.5101 | Public | 13650.9079 | All Other End Locations | 13404.6481 |
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | All Other End Locations | 36939.2788 |
| TEXAS | 123400.5101 | Trained Spotter | 13997.7124 | All Other End Locations | 13576.2724 |
| KANSAS | 87771.2355000001 | | | All Other End Locations | 24891.0836 |
| TEXAS | 123400.5101 | | | All Other End Locations | 58523.2932000001 |
| All Other States | 1149279.5923 | | | All Other End Locations | 1149279.5923 |

The following query shows the same results for the first level used in the example above.

Run the query

Kusto

```
StormEvents
| where State !in ('TEXAS', 'KANSAS')
| summarize sum(BeginLat)
```

**Output**

| sum_BeginLat |
|---|
| 1149279.5923 |

Request another column (EventType) to the top-nested result.

Run the query

Kusto

```
StormEvents
| top-nested 2 of State       by sum(BeginLat),
```

```
  top-nested 2 of Source      by sum(BeginLat),
  top-nested 1 of EndLocation by sum(BeginLat),
  top-nested   of EventType   by tmp = max(1)
| project-away tmp
```

**Output**

| State | aggregated_State | Source | aggregated_Source | EndLocation | aggregated_EndLocation | EventType |
|---|---|---|---|---|---|---|
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | SHARON SPGS | 388.7404 | Thunderstorm Wind |
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | SHARON SPGS | 388.7404 | Hail |
| KANSAS | 87771.2355000001 | Trained Spotter | 21279.7083 | SHARON SPGS | 388.7404 | Tornado |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | BUCKLIN | 488.2457 | Hail |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | BUCKLIN | 488.2457 | Thunderstorm Wind |
| KANSAS | 87771.2355000001 | Public | 22855.6206 | BUCKLIN | 488.2457 | Flood |
| TEXAS | 123400.5101 | Trained Spotter | 13997.7124 | CLAUDE | 421.44 | Hail |
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | PERRYTON | 289.3178 | Hail |
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | PERRYTON | 289.3178 | Flood |
| TEXAS | 123400.5101 | Law Enforcement | 37228.5966 | PERRYTON | 289.3178 | Flash Flood |

Give an index sort order for each value in this level (per group) to sort the result by the last nested level (in this example by EndLocation):

Run the query

Kusto

```
StormEvents
| top-nested 2 of State  by sum(BeginLat),    top-nested 2 of Source by sum(BeginLat),    top-
nested 4 of EndLocation by  sum(BeginLat)
| order by State , Source, aggregated_EndLocation
| summarize EndLocations = make_list(EndLocation, 10000) , endLocationSums =
make_list(aggregated_EndLocation, 10000) by State, Source
| extend indicies = range(0, array_length(EndLocations) - 1, 1)
| mv-expand EndLocations, endLocationSums, indicies
```

**Output**

| State | Source | EndLocations | endLocationSums | indices |
|---|---|---|---|---|
| TEXAS | Trained Spotter | CLAUDE | 421.44 | 0 |
| TEXAS | Trained Spotter | AMARILLO | 316.8892 | 1 |

| State | Source | EndLocations | endLocationSums | indices |
|---|---|---|---|---|
| TEXAS | Trained Spotter | DALHART | 252.6186 | 2 |
| TEXAS | Trained Spotter | PERRYTON | 216.7826 | 3 |
| TEXAS | Law Enforcement | PERRYTON | 289.3178 | 0 |
| TEXAS | Law Enforcement | LEAKEY | 267.9825 | 1 |
| TEXAS | Law Enforcement | BRACKETTVILLE | 264.3483 | 2 |
| TEXAS | Law Enforcement | GILMER | 261.9068 | 3 |
| KANSAS | Trained Spotter | SHARON SPGS | 388.7404 | 0 |
| KANSAS | Trained Spotter | ATWOOD | 358.6136 | 1 |
| KANSAS | Trained Spotter | LENORA | 317.0718 | 2 |
| KANSAS | Trained Spotter | SCOTT CITY | 307.84 | 3 |
| KANSAS | Public | BUCKLIN | 488.2457 | 0 |
| KANSAS | Public | ASHLAND | 446.4218 | 1 |
| KANSAS | Public | PROTECTION | 446.11 | 2 |
| KANSAS | Public | MEADE STATE PARK | 371.1 | 3 |

The following example returns the two most-recent events for each US state, with some information per event. Note the use of the `max(1)` (which is then projected away) for columns which just require propagation through the operator without any selection logic.

Run the query

Kusto

```
StormEvents
| top-nested of State by Ignore0=max(1),
  top-nested 2 of StartTime by Ignore1=max(StartTime),
  top-nested of EndTime by Ignore2=max(1),
  top-nested of EpisodeId by Ignore3=max(1)
| project-away Ignore*
| order by State asc, StartTime desc
```

## Retrieve the latest records per identity

If you have a table with an ID column and a timestamp column, you can use the top-nested operator to query the latest two records for each unique value of ID. The latest records are defined by the highest value of timestamp.

Run the query

Kusto

```
datatable(id: string, timestamp: datetime, otherInformation: string)
[
    "Barak", datetime(2015-01-01), "1",
```

```
    "Barak", datetime(2016-01-01), "2",
    "Barak", datetime(2017-01-20), "3",
    "Donald", datetime(2017-01-20), "4",
    "Donald", datetime(2017-01-18), "5",
    "Donald", datetime(2017-01-19), "6"
]
| top-nested of id by dummy0=max(1),
top-nested 2 of timestamp by dummy1=max(timestamp),
top-nested of otherInformation by dummy2=max(1)
| project-away dummy0, dummy1, dummy2
```

**Output**

| id | timestamp | otherInformation |
|---|---|---|
| Barak | 2016-01-01T00:00:00Z | 2 |
| Donald | 2017-01-19T00:00:00Z | 6 |
| Barak | 2017-01-20T00:00:00Z | 3 |
| Donald | 2017-01-20T00:00:00Z | 4 |

Here's a step-by-step explanation of the query:

1. The `datatable` creates a test dataset.
2. The first `top-nested` clause returns all distinct values of `id`.
3. The second `top-nested` clause selects the top two records with the highest `timestamp` for each id.
4. The third `top-nested` clause adds the `otherInformation` column for each record.
5. The `project-away` operator removes the dummy columns introduced by the top-nested operator.

---

# Feedback

# top-hitters operator

Article • 03/23/2023

Returns an approximation for the most popular distinct values, or the values with the largest sum, in the input.

> **Note**
>
> `top-hitters` uses an approximation algorithm optimized for performance when the input data is large. The approximation is based on the **Count-Min-Sketch** algorithm.

## Syntax

*T* `|` `top-hitters` *NumberOfValues* `of` *ValueExpression* [ `by` *SummingExpression* ]

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | ✓ | The input tabular expression. |
| *NumberOfValues* | int, long, or real | ✓ | The number of distinct values of *ValueExpression*. |
| *ValueExpression* | string | ✓ | An expression over the input table *T* whose distinct values are returned. |
| *SummingExpression* | string | | If specified, a numeric expression over the input table *T* whose sum per distinct value of *ValueExpression* establishes which values to emit. If not specified, the count of each distinct value of *ValueExpression* is used instead. |

## Remarks

The first syntax (no *SummingExpression*) is conceptually equivalent to:

*T* `|` `summarize` `C``=``count()` `by` *ValueExpression* `|` `top` *NumberOfValues* by `C` `desc`

The second syntax (with *SummingExpression*) is conceptually equivalent to:

*T* `|` `summarize` `S``=``sum(*SummingExpression*)` `by` *ValueExpression* `|` `top` *NumberOfValues* by `S` `desc`

# Examples

## Get most frequent items

The next example shows how to find top-5 types of storms.

Run the query

```
Kusto
```

```
StormEvents
| top-hitters 5 of EventType
```

**Output**

| EventType | approximate_count_EventType |
|---|---|
| Thunderstorm Wind | 13015 |
| Hail | 12711 |
| Flash Flood | 3688 |
| Drought | 3616 |
| Winter Weather | 3349 |

## Get top hitters based on column value

The next example shows how to find the States with the most "Thunderstorm Wind" events.

Run the query

```
Kusto
```

```
StormEvents
| where EventType == "Thunderstorm Wind"
| top-hitters 10 of State
```

### Output

| State | approximate_sum_State |
|---|---|
| TEXAS | 830 |
| GEORGIA | 609 |
| MICHIGAN | 602 |
| IOWA | 585 |
| PENNSYLVANIA | 549 |
| ILLINOIS | 533 |
| NEW YORK | 502 |
| VIRGINIA | 482 |
| KANSAS | 476 |
| OHIO | 455 |

# Feedback

**Was this page helpful?** Yes No

Provide product feedback | Get help at Microsoft Q&A

# union operator

Article • 03/12/2023

Takes two or more tables and returns the rows of all of them.

## Syntax

[ *T* `|` ] `union` [ *UnionParameters* ] [`kind=` `inner`|`outer`] [`withsource=` *ColumnName*] [`isfuzzy=` `true`|`false`] *Tables*

> ⓘ **Note**
>
> The operation of the `union` operator can be altered by setting the `best_effort` request property to `true`, using either a **set statement** or through **client request properties**. When this property is set to `true`, the `union` operator will disregard fuzzy resolution and connectivity failures to execute any of the sub-expressions being "unioned" and yield a warning in the query status results.

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | | The input tabular expression. |
| *UnionParameters* | string | | Zero or more space-separated parameters in the form of *Name* `=` *Value* that control the behavior of the row-match operation and execution plan. See supported union parameters. |

| Name | Type | Required | Description |
|---|---|---|---|
| `kind` | string | | Either `inner` or `outer`. `inner` causes the result to have the subset of columns that are common to all of the input tables. `outer` causes the result to have all the columns that occur in any of the inputs. Cells that aren't defined by an input row are set to `null`. The default is `outer`.<br><br>With `outer`, the result has all the columns that occur in any of the inputs, one column for each name and type occurrences. This means that if a column appears in multiple tables and has multiple types, it has a corresponding column for each type in the union's result. This column name is suffixed with a '_' followed by the origin column type. |
| `withsource=`*ColumnName* | string | | If specified, the output includes a column called *ColumnName* whose value indicates which source table has contributed each row. If the query effectively references tables from more than one database including the default database, then the value of this column has a table name qualified with the database. **cluster and database** qualifications are present in the value if more than one cluster is referenced. |
| `isfuzzy` | bool | | If set to `true`, allows fuzzy resolution of union legs. The set of union sources is reduced to the set of table references that exist and are accessible at the time while analyzing the query and preparing for execution. If at least one such table was found, any resolution failure yields a warning in the query status results, but won't prevent the query execution. If no resolutions were successful, the query returns an error. The default is `false`.<br><br>`isfuzzy=true` only applies to the `union` sources resolution phase. Once the set of source tables is determined, possible additional query failures won't be suppressed. |

| Name | Type | Required | Description |
|---|---|---|---|
| *Tables* | string | | One or more comma-separated table references, a query expression enclosed with parenthesis, or 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 `E`. |

## Supported union parameters

| Name | Type | Required | Description |
|---|---|---|---|
| `hint.concurrency` | int | | Hints the system how many concurrent subqueries of the `union` operator should be executed in parallel. The default is the number of CPU cores on the single node of the cluster (2 to 16). |
| `hint.spread` | int | | Hints the system how many nodes should be used by the concurrent `union` subqueries execution. The default is 1. |

> ⓘ Note
>
> - The `union` scope can include **let statements** if attributed with the `view` keyword.
> - The `union` scope will not include **functions**. To include a function, define a **let statement** with the `view` keyword.
> - There's no guarantee of the order in which the union legs will appear, but if each leg has an `order by` operator, then each leg will be sorted.

# Returns

A table with as many rows as there are in all the input tables.

# Examples

## Tables with string in name or column

```
Kusto
```

```
union K* | where * has "Kusto"
```

Rows from all tables in the database whose name starts with `K`, and in which any column includes the word `Kusto`.

## Distinct count

Kusto

```
union withsource=SourceTable kind=outer Query, Command
| where Timestamp > ago(1d)
| summarize dcount(UserId)
```

The number of distinct users that have produced either a `Query` event or a `Command` event over the past day. In the result, the 'SourceTable' column will indicate either "Query" or "Command".

Kusto

```
Query
| where Timestamp > ago(1d)
| union withsource=SourceTable kind=outer
   (Command | where Timestamp > ago(1d))
| summarize dcount(UserId)
```

This more efficient version produces the same result. It filters each table before creating the union.

## Using `isfuzzy=true`

Kusto

```
// Using union isfuzzy=true to access non-existing view:
let View_1 = view () { print x=1 };
let View_2 = view () { print x=1 };
let OtherView_1 = view () { print x=1 };
union isfuzzy=true
(View_1 | where x > 0),
(View_2 | where x > 0),
(View_3 | where x > 0)
| count
```

**Output**

| Count |
| --- |
| 2 |

Observing Query Status - the following warning returned: `Failed to resolve entity 'View_3'`

Kusto

```
// Using union isfuzzy=true and wildcard access:
let View_1 = view () { print x=1 };
let View_2 = view () { print x=1 };
let OtherView_1 = view () { print x=1 };
union isfuzzy=true View*, SomeView*, OtherView*
| count
```

**Output**

| Count |
| --- |
| 3 |

Observing Query Status - the following warning returned: `Failed to resolve entity 'SomeView*'`

## Source columns types mismatch

Kusto

```
let View_1 = view () { print x=1 };
let View_2 = view () { print x=toint(2) };
union withsource=TableName View_1, View_2
```

**Output**

| TableName | x_long | x_int |
| --- | --- | --- |
| View_1 | 1 | |
| View_2 | | 2 |

Kusto

```
let View_1 = view () { print x=1 };
let View_2 = view () { print x=toint(2) };
```

```
let View_3 = view () { print x_long=3 };
union withsource=TableName View_1, View_2, View_3
```

**Output**

| TableName | x_long1 | x_int | x_long |
|---|---|---|---|
| View_1 | 1 | | |
| View_2 | | 2 | |
| View_3 | | | 3 |

Column `x` from `View_1` received the suffix `_long`, and as a column named `x_long` already exists in the result schema, the column names were de-duplicated, producing a new column- `x_long1`

---

# Feedback

# where operator

Article • 01/16/2023

Filters a table to the subset of rows that satisfy a predicate.

> The `where` and `filter` operators are equivalent

## Syntax

*T* `| where` *Predicate*

## Parameters

| Name | Type | Required | Description |
|---|---|---|---|
| *T* | string | ✓ | Tabular input whose records are to be filtered. |
| *Predicate* | string | ✓ | Expression that evaluates to a bool for each row in *T*. |

## Returns

Rows in *T* for which *Predicate* is `true`.

> ⓘ Note
>
> All filtering functions return false when compared with null values. Use special null-aware functions to write queries that handle null values.
>
> - isnull()
> - isnotnull()
> - isempty()
> - isnotempty()

## Performance tips

- **Use simple comparisons** between column names and constants. ('Constant' means constant over the table - so `now()` and `ago()` are OK, and so are scalar values assigned using a let statement.)

For example, prefer `where Timestamp >= ago(1d)` to `where bin(Timestamp, 1d) == ago(1d)`.

- **Simplest terms first**: If you have multiple clauses conjoined with `and`, put first the clauses that involve just one column. So `Timestamp > ago(1d) and OpId == EventId` is better than the other way around.

For more information, see the summary of available String operators and the summary of available Numerical operators.

# Examples

## Order comparisons by complexity

The following query returns storm records that report damaged property, are floods, and start and end in different places.

Notice that we put the comparison between two columns last, as the where operator can't use the index and forces a scan.

Run the query

Kusto

```
StormEvents
| project DamageProperty, EventType, BeginLocation, EndLocation
| where DamageProperty > 0
    and EventType == "Flood"
    and BeginLocation != EndLocation
```

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

| DamageProperty | EventType | BeginLocation | EndLocation |
|---|---|---|---|
| 5000 | Flood | FAYETTE CITY LOWBER | |
| 5000 | Flood | MORRISVILLE WEST WAYNESBURG | |
| 10000 | Flood | COPELAND HARRIS GROVE | |
| 5000 | Flood | GLENFORD MT PERRY | |
| 25000 | Flood | EAST SENECA BUFFALO AIRPARK ARPT | |
| 20000 | Flood | EBENEZER SLOAN | |

| DamageProperty | EventType | BeginLocation | EndLocation |
|---|---|---|---|
| 10000 | Flood | BUEL CALHOUN | |
| 10000 | Flood | GOODHOPE WEST MILFORD | |
| 5000 | Flood | DUNKIRK FOREST | |
| 20000 | Flood | FARMINGTON MANNINGTON | |

## Check if column contains string

The following query returns the rows in which the word "cow" appears in any column.

Run the query

Kusto

```
StormEvents
| where * has "cow"
```

# Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A