Window functions overview

Article • 03/02/2023

Window functions operate on multiple rows (records) in a row set at a time. Unlike aggregation functions, window functions require that the rows in the row set be serialized (have a specific order to them). Window functions may depend on the order to determine the result.

Window functions can only be used on serialized sets. The easiest way to serialize a row set is to use the serialize operator. This operator "freezes" the order of rows in an arbitrary manner. If the order of serialized rows is semantically important, use the sort operator to force a particular order.

The serialization process has a non-trivial cost associated with it. For example, it might prevent query parallelism in many scenarios. Therefore, don't apply serialization unnecessarily. If necessary, rearrange the query to perform serialization on the smallest row set possible.

Serialized row set

An arbitrary row set (such as a table, or the output of a tabular operator) can be serialized in one of the following ways:

1. By sorting the row set. See below for a list of operators that emit sorted row sets.
2. By using the serialize operator.

Many tabular operators serialize output whenever the input is already serialized, even if the operator doesn't itself guarantee that the result is serialized. For example, this property is guaranteed for the extend operator, the project operator, and the where operator.

Operators that emit serialized row sets by sorting

- sort operator
- top operator
- top-hitters operator

Operators that preserve the serialized row set property

- extend operator
- mv-expand operator
- parse operator
- project operator
- project-away operator
- project-rename operator
- take operator
- where operator

Feedback

next()

Article • 03/23/2023

Returns the value of a column in a row that is at some offset following the current row in a serialized row set.

Syntax

`next(`*column*`,` [*offset*`,` *default_value*]`)`

Parameters

Name	Type	Required	Description
column	string	✓	The column from which to get the values.
offset	int		The amount of rows to move from the current row. Default is 1.
default_value	scalar		The default value when there's no value in the next row. When no default value is specified, `null` is used.

Examples

Filter data based on comparison between adjacent rows

The following query returns rows that show breaks longer than a quarter of a second between calls to `sensor-9`.

Run the query

Kusto

```
TransformedSensorsData
| where SensorName == 'sensor-9'
| sort by Timestamp asc
| extend timeDiffInMilliseconds = datetime_diff('millisecond',
next(Timestamp, 1), Timestamp)
| where timeDiffInMilliseconds > 250
```

Output

Timestamp	SensorName	Value	PublisherId	MachineId	timeDiff
2022-04-13T00:58:53.048506Z	sensor-9	0.39217481975439894	fdbd39ab-82ac-4ca0-99ed-2f83daf3f9bb	M100	251
2022-04-13T01:07:09.63713Z	sensor-9	0.46645392778288297	e3ed081e-501b-4d59-8e60-8524633d9131	M100	313
2022-04-13T01:07:10.858267Z	sensor-9	0.693091598493419	278ca033-2b5e-4f2c-b493-00319b275aea	M100	254
2022-04-13T01:07:11.203834Z	sensor-9	0.52415808840249778	4ea27181-392d-4947-b811-ad5af02a54bb	M100	331
2022-04-13T01:07:14.431908Z	sensor-9	0.35430645405452	0af415c2-59dc-4a50-89c3-9a18ae5d621f	M100	268
...	...	...	...	...	...

Perform aggregation based on comparison between adjacent rows

The following query calculates the average time difference in milliseconds between calls to `sensor-9`.

Run the query

Kusto

```
TransformedSensorsData
| where SensorName == 'sensor-9'
| sort by Timestamp asc
| extend timeDiffInMilliseconds = datetime_diff('millisecond',
next(Timestamp, 1), Timestamp)
| summarize avg(timeDiffInMilliseconds)
```

Output

avg_timeDiffInMilliseconds
30.726900061254298

Extend row with data from the next row

In the following query, as part of the serialization done with the serialize operator, a new column `next_session_type` is added with data from the next row.

Run the query

Kusto

```
ConferenceSessions
| where conference == 'Build 2019'
| serialize next_session_type = next(session_type)
| project time_and_duration, session_title, session_type, next_session_type
```

Output

time_and_duration	session_title	session_type	next_session_type
Mon, May 6, 8:30-10:00 am	Vision Keynote - Satya Nadella	Keynote	Expo Session
Mon, May 6, 1:20-1:40 pm	Azure Data Explorer: Advanced Time Series analysis	Expo Session	Breakout
Mon, May 6, 2:00-3:00 pm	Azure's Data Platform - Powering Modern Applications and Cloud Scale Analytics at Petabyte Scale	Breakout	Expo Session
Mon, May 6, 4:00-4:20 pm	How BASF is using Azure Data Services	Expo Session	Expo Session
Mon, May 6, 6:50 - 7:10 pm	Azure Data Explorer: Operationalize your ML models	Expo Session	Expo Session
...	...	...	...

Feedback

prev()

Article • 03/23/2023

Returns the value of a specific column in a specified row. The specified row is at a specified offset from the current row in a serialized row set.

Syntax

`prev(`*column*`,` [*offset*]`,` [*default_value*] `)`

Parameters

Name	Type	Required	Description
column	string	✓	The column from which to get the values.
offset	int		The offset to go back in rows. The default is 1.
default_value	scalar		The default value to be used when there are no previous rows from which to take the value. The default is `null`.

Examples

Filter data based on comparison between adjacent rows

The following query returns rows that show breaks longer than a quarter of a second between calls to `sensor-9`.

Run the query

Kusto

```
TransformedSensorsData
| where SensorName == 'sensor-9'
| sort by Timestamp asc
| extend timeDiffInMilliseconds = datetime_diff('millisecond', Timestamp,
prev(Timestamp, 1))
| where timeDiffInMilliseconds > 250
```

Output

Timestamp	SensorName	Value	PublisherId	MachineId	timeDiff
2022-04-13T00:58:53.048506Z	sensor-9	0.39217481975439894	fdbd39ab-82ac-4ca0-99ed-2f83daf3f9bb	M100	251
2022-04-13T01:07:09.63713Z	sensor-9	0.46645392778288297	e3ed081e-501b-4d59-8e60-8524633d9131	M100	313
2022-04-13T01:07:10.858267Z	sensor-9	0.693091598493419	278ca033-2b5e-4f2c-b493-00319b275aea	M100	254
2022-04-13T01:07:11.203834Z	sensor-9	0.52415808840249778	4ea27181-392d-4947-b811-ad5af02a54bb	M100	331
2022-04-13T01:07:14.431908Z	sensor-9	0.35430645405452	0af415c2-59dc-4a50-89c3-9a18ae5d621f	M100	268
...	...	...	...	...	...

Perform aggregation based on comparison between adjacent rows

The following query calculates the average time difference in milliseconds between calls to `sensor-9`.

Run the query

Kusto

```
TransformedSensorsData
| where SensorName == 'sensor-9'
| sort by Timestamp asc
| extend timeDiffInMilliseconds = datetime_diff('millisecond', Timestamp,
prev(Timestamp, 1))
| summarize avg(timeDiffInMilliseconds)
```

Output

avg_timeDiffInMilliseconds
30.726900061254298

Extend row with data from the previous row

In the following query, as part of the serialization done with the serialize operator, a new column `previous_session_type` is added with data from the previous row. Since there was no session prior to the first session, the column is empty in the first row.

Run the query

Kusto

```
ConferenceSessions
| where conference == 'Build 2019'
| serialize previous_session_type = prev(session_type)
| project time_and_duration, session_title, session_type,
previous_session_type
```

Output

time_and_duration	session_title	session_type	previous_session_type
Mon, May 6, 8:30-10:00 am	Vision Keynote - Satya Nadella	Keynote	
Mon, May 6, 1:20-1:40 pm	Azure Data Explorer: Advanced Time Series analysis	Expo Session	Keynote
Mon, May 6, 2:00-3:00 pm	Azure's Data Platform - Powering Modern Applications and Cloud Scale Analytics at Petabyte Scale	Breakout	Expo Session
Mon, May 6, 4:00-4:20 pm	How BASF is using Azure Data Services	Expo Session	Breakout
Mon, May 6, 6:50 - 7:10 pm	Azure Data Explorer: Operationalize your ML models	Expo Session	Expo Session
...	...	...	...

Feedback

row_cumsum()

Article • 01/31/2023

Calculates the cumulative sum of a column in a serialized row set.

Syntax

`row_cumsum(` *term* [`,` *restart*] `)`

Parameters

Name	Type	Required	Description
term	int, long, or real	✓	The expression indicating the value to be summed.
restart	bool		Indicates when the accumulation operation should be restarted, or set back to 0. It can be used to indicate partitions in the data.

Returns

The function returns the cumulative sum of its argument.

Examples

The following example shows how to calculate the cumulative sum of the first few even integers.

Run the query

Kusto

```
datatable (a:long) [
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
]
| where a%2==0
| serialize cs=row_cumsum(a)
```

a	cs

a	cs
2	2
4	6
6	12
8	20
10	30

This example shows how to calculate the cumulative sum (here, of `salary`) when the data is partitioned (here, by `name`):

Run the query

Kusto

```
datatable (name:string, month:int, salary:long)
[
    "Alice", 1, 1000,
    "Bob",   1, 1000,
    "Alice", 2, 2000,
    "Bob",   2, 1950,
    "Alice", 3, 1400,
    "Bob",   3, 1450,
]
| order by name asc, month asc
| extend total=row_cumsum(salary, name != prev(name))
```

name	month	salary	total
Alice	1	1000	1000
Alice	2	2000	3000
Alice	3	1400	4400
Bob	1	1000	1000
Bob	2	1950	2950
Bob	3	1450	4400

Feedback

row_number()

Article • 01/31/2023

Returns the current row's index in a serialized row set.

The row index starts by default at `1` for the first row, and is incremented by `1` for each additional row. Optionally, the row index can start at a different value than `1`. Additionally, the row index may be reset according to some provided predicate.

Syntax

`row_number(` [*StartingIndex* [`,` *Restart*]] `)`

Name	Type	Required	Description
StartingIndex	long		The value of the row index to start at or restart to. The default value is 1.
restart	bool		Indicates when the numbering is to be restarted to the *StartingIndex* value. The default is `false`.

Returns

The function returns the row index of the current row as a value of type `long`.

Examples

The following example returns a table with two columns, the first column (`a`) with numbers from `10` down to `1`, and the second column (`rn`) with numbers from `1` up to `10`:

Run the query

Kusto

```
range a from 1 to 10 step 1
| sort by a desc
| extend rn=row_number()
```

The following example is similar to the above, only the second column (`rn`) starts at `7`:

Run the query

```
range a from 1 to 10 step 1
| sort by a desc
| extend rn=row_number(7)
```

The last example shows how one can partition the data and number the rows per each partition. Here, we partition the data by `Airport`:

Run the query

```
datatable (Airport:string, Airline:string, Departures:long)
[
  "TLV", "LH", 1,
  "TLV", "LY", 100,
  "SEA", "LH", 1,
  "SEA", "BA", 2,
  "SEA", "LY", 0
]
| sort by Airport asc, Departures desc
| extend Rank=row_number(1, prev(Airport) != Airport)
```

Running this query produces the following result:

Airport	Airline	Departures	Rank
SEA	BA	2	1
SEA	LH	1	2
SEA	LY	0	3
TLV	LY	100	1
TLV	LH	1	2

Feedback

row_rank_dense()

Article • 06/07/2023

Returns the current row's dense rank in a serialized row set.

The row rank starts by default at `1` for the first row, and is incremented by `1` whenever the provided *Term* is different than the previous row's *Term*.

Syntax

`row_rank_dense` `(` *Term* `)`

Parameters

Name	Type	Required	Description
Term	string	✓	An expression indicating the value to consider for the rank. The rank is increased whenever the *Term* changes.
restart	bool		Indicates when the numbering is to be restarted to the *StartingIndex* value. The default is `false`.

Returns

Returns the row rank of the current row as a value of type `long`.

Example

The following query shows how to rank the `Airline` by the number of departures from the SEA `Airport` using dense rank.

Run the query

Kusto

```
datatable (Airport:string, Airline:string, Departures:long)
[
  "SEA", "LH", 3,
  "SEA", "LY", 100,
  "SEA", "UA", 3,
  "SEA", "BA", 2,
```

```
  "SEA", "EL", 3
]
| sort by Departures asc
| extend Rank=row_rank_dense(Departures)
```

Output

Airport	Airline	Departures	Rank
SEA	BA	2	1
SEA	LH	3	2
SEA	UA	3	2
SEA	EL	3	2
SEA	LY	100	3

Run the query

The following example shows how to rank the `Airline` by the number of departures per each partition. Here, we partition the data by `Airport`:

Kusto

```
datatable (Airport:string, Airline:string, Departures:long)
[
  "SEA", "LH", 3,
  "SEA", "LY", 100,
  "SEA", "UA", 3,
  "SEA", "BA", 2,
  "SEA", "EL", 3,
  "AMS", "EL", 1,
  "AMS", "BA", 1
]
| sort by Airport desc, Departures asc
| extend Rank=row_rank_dense(Departures, prev(Airport) != Airport)
```

Output

Airport	Airline	Departures	Rank
SEA	BA	2	1
SEA	LH	3	2
SEA	UA	3	2

Airport	Airline	Departures	Rank
SEA	EL	3	2
SEA	LY	100	3
AMS	EL	1	1
AMS	BA	1	1

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

row_rank_min()

Article • 06/07/2023

Returns the current row's minimal rank in a serialized row set.

The rank is the minimal row number that the current row's *Term* appears in.

Syntax

`row_rank_min` `(` *Term* `)`

Parameters

Name	Type	Required	Description
Term	string	✓	An expression indicating the value to consider for the rank. The rank is the minimal row number for *Term*.
restart	bool		Indicates when the numbering is to be restarted to the *StartingIndex* value. The default is `false`.

Returns

Returns the row rank of the current row as a value of type `long`.

Example

The following query shows how to rank the `Airline` by the number of departures from the SEA `Airport`.

Run the query

Kusto

```
datatable (Airport:string, Airline:string, Departures:long)
[
  "SEA", "LH", 3,
  "SEA", "LY", 100,
  "SEA", "UA", 3,
  "SEA", "BA", 2,
  "SEA", "EL", 3
]
```

```
| sort by Departures asc
| extend Rank=row_rank_min(Departures)
```

Output

Airport	Airline	Departures	Rank
SEA	BA	2	1
SEA	LH	3	2
SEA	UA	3	2
SEA	EL	3	2
SEA	LY	100	5

Feedback

row_window_session()

Article • 01/23/2023

Calculates session start values of a column in a serialized row set.

Syntax

`row_window_session` `(` *`Expr`* `,` *`MaxDistanceFromFirst`* `,` *`MaxDistanceBetweenNeighbors`* [`,` *`Restart`*] `)`

- *`Expr`* is an expression whose values are grouped together in sessions. Null values produce null values, and the next value starts a new session. *`Expr`* must be a scalar expression of type `datetime`.
- *`MaxDistanceFromFirst`* establishes one criterion for starting a new session: The maximum distance between the current value of *`Expr`* and the value of *`Expr`* at the beginning of the session. It's a scalar constant of type `timespan`.
- *`MaxDistanceBetweenNeighbors`* establishes a second criterion for starting a new session: The maximum distance from one value of *`Expr`* to the next. It's a scalar constant of type `timespan`.
- *Restart* is an optional scalar expression of type `boolean`. If specified, every value that evaluates to `true` will immediately restart the session.

Returns

The function returns the values at the beginning of each session.

The function has the following conceptual calculation model:

1. Goes over the input sequence of *`Expr`* values in order.
2. For every value, determines if it establishes a new session.
3. If it establishes a new session, it emits the value of *`Expr`*. Otherwise, emits the previous value of *`Expr`*.

> ⓘ Note

The condition that determines if the value represents a new session is a logical OR one of the following conditions:

- If there was no previous session value, or the previous session value was null.
- If the value of *Expr* equals or exceeds the previous session value plus *MaxDistanceFromFirst*.
- If the value of *Expr* equals or exceeds the previous value of *Expr* plus *MaxDistanceBetweenNeighbors*.
- If *Restart* condition is specified and evaluates to `true`.

Examples

The following example shows how to calculate the session start values for a table with two columns: an `ID` column that identifies a sequence, and a `Timestamp` column that gives the time at which each record occurred. In this example, a session can't exceed 1 hour, and it continues as long as records are less than 5 minutes apart.

Kusto

```
datatable (ID:string, Timestamp:datetime) [
    // ...
]
| sort by ID asc, Timestamp asc
| extend SessionStarted = row_window_session(Timestamp, 1h, 5m, ID !=
prev(ID))
```

See also

- scan operator

Feedback

Query consistency

Article • 05/31/2023

Query consistency refers to how queries and updates are synchronized. There are two supported modes of query consistency:

- Strong consistency: Strong consistency ensures immediate access to the most recent updates, such as data appends, deletions, and schema modifications. Strong consistency is the default consistency mode. Due to synchronization, this consistency mode performs slightly less well than weak consistency mode in terms of concurrency.
- Weak consistency: With weak consistency, there may be a delay before query results reflect the latest database updates. Typically, this delay ranges from 1 to 2 minutes. Weak consistency can support higher query concurrency rates than strong consistency.

For example, if 1000 records are ingested each minute into a table in the database, queries over that table running with strong consistency will have access to the most-recently ingested records, whereas queries over that table running with weak consistency may not have access to some of records from the last few minutes.

> ⓘ **Note**
>
> By default, queries run with strong consistency. We recommend only switching to weak consistency when necessary for supporting higher query concurrency.

Use cases for strong consistency

If you have a strong dependency on updates that occurred in the database in the last few minutes, use strong consistency.

For example, the following query counts the number of error records in the 5 minutes and triggers an alert that count is larger than 0. This use case is best handled with strong consistency, since your insights may be altered you don't have access to records ingested in the past few minutes, as may be the case with weak consistency.

Kusto

```
my_table
| where timestamp between(ago(5m)..now())
```

```
| where level == "error"
| count
```

In addition, strong consistency should be used when database metadata is large. For instance. there are millions of data extents in the database, using weak consistency would result in query heads downloading and deserializing extensive metadata artifacts from persistent storage, which may increase the likelihood of transient failures in downloads and related operations.

Use cases for weak consistency

If you don't have a strong dependency on updates that occurred in the database in the last few minutes, and you need high query concurrency, use weak consistency.

For example, the following query counts the number of error records per week in the last 90 days. Weak consistency is appropriate in this case, since your insights are unlikely to be impacted records ingested in the past few minutes are omitted.

Kusto

```
my_table
| where timestamp between(ago(90d) .. now())
| where level == "error"
| summarize count() by level, startofweek(Timestamp)
```

Weak consistency modes

The following table summarizes the four modes of weak query consistency.

Mode	Description
Random	Queries are routed randomly to one of the nodes in the cluster that can serve as a weakly consistent query head.
Affinity by database	Queries within the same database are routed to the same weakly consistent query head, ensuring consistent execution for that database.
Affinity by query text	Queries with the same query text hash are routed to the same weakly consistent query head, which is beneficial for leveraging query caching.
Affinity by session ID	Queries with the same session ID hash are routed to the same weakly consistent query head, ensuring consistent execution within a session.

Affinity by database

The affinity by database mode ensures that queries running against the same database are executed against the same version of the database, although not necessarily the most recent version of the database. This mode is useful when ensuring consistent execution within a specific database is important. However. there's an imbalance in the number of queries across databases, then this mode may result in uneven load distribution.

Affinity by query text

The affinity by query text mode is beneficial when queries leverage the Query results cache. This mode routes repeating queries frequently executed by the same identity to the same query head, allowing them to benefit from cached results and reducing the load on the cluster.

Affinity by session ID

The affinity by session ID mode ensures that queries belonging to the same user activity or session are executed against the same version of the database, although not necessarily the most recent one. To use this mode, the session ID needs to be explicitly specified in each query's client request properties. This mode is helpful in scenarios where consistent execution within a session is essential.

How to specify query consistency

You can specify the query consistency mode by the client sending the request or using a server side policy. If it isn't specified by either, the default mode of strong consistency applies.

- Client sending the request: Use the `queryconsistency` client request property. This method sets the query consistency mode for a specific query and doesn't affect the overall effective consistency mode, which is determined by the default or the server-side policy. For more information, see client request properties.
- Server side policy: Use the `QueryConsistency` property of the Query consistency policy. This method sets the query consistency mode at the workload group level, which eliminates the need for users to specify the consistency mode in their client request properties and allows for enforcing desired consistency modes. For more information, see Query consistency policy.

> **Note**
>
> If using the **Kusto .NET SDK**, you can set the query consistency through the connection string. This setting will apply to all queries sent through that particular connection string. For more information, see **Connection string properties**.

Next steps

- To customize parameters for queries running with weak consistency, use the Query weak consistency policy.

Feedback