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

Query limits

Article • 03/01/2023

Kusto is an ad-hoc query engine that hosts large data sets and attempts to satisfy queries by holding all relevant data in-memory. There's an inherent risk that queries will monopolize the service resources without bounds. Kusto provides several built-in protections in the form of default query limits. If you're considering removing these limits, first determine whether you actually gain any value by doing so.

Limit on request concurrency

Request concurrency is a limit that a cluster imposes on several requests running at the same time.

- The default value of the limit depends on the SKU the cluster is running on, and is calculated as: `Cores-Per-Node x 10`.
  - For example, for a cluster that's set up on D14v2 SKU, where each machine has 16 vCores, the default limit is `16 cores x10 = 160`.
- The default value can be changed by configuring the request rate limit policy of the `default` workload group.
  - The actual number of requests that can run concurrently on a cluster depends on various factors. The most dominant factors are cluster SKU, cluster's available resources, and usage patterns. The policy can be configured based on load tests performed on production-like usage patterns.

For more information, see Optimize for high concurrency with Azure Data Explorer.

Limit on result set size (result truncation)

Result truncation is a limit set by default on the result set returned by the query. Kusto limits the number of records returned to the client to **500,000**, and the overall data size for those records to **64 MB**. When either of these limits is exceeded, the query fails with a "partial query failure". Exceeding overall data size will generate an exception with the message:

```
The Kusto DataEngine has failed to execute a query: 'Query result set has
exceeded the internal data size limit 67108864
(E_QUERY_RESULT_SET_TOO_LARGE).'
```

Exceeding the number of records will fail with an exception that says:

```
The Kusto DataEngine has failed to execute a query: 'Query result set has
exceeded the internal record count limit 500000
(E_QUERY_RESULT_SET_TOO_LARGE).'
```

There are several strategies for dealing with this error.

- Reduce the result set size by modifying the query to only return interesting data. This strategy is useful when the initial failing query is too "wide". For example, the query doesn't project away data columns that aren't needed.
- Reduce the result set size by shifting post-query processing, such as aggregations, into the query itself. The strategy is useful in scenarios where the output of the query is fed to another processing system, and that then does other aggregations.
- Switch from queries to using data export when you want to export large sets of data from the service.
- Instruct the service to suppress this query limit using `set` statements listed below or flags in client request properties.

Methods for reducing the result set size produced by the query include:

- Use the summarize operator group and aggregate over similar records in the query output. Potentially sample some columns by using the take_any aggregation function.
- Use a take operator to sample the query output.
- Use the substring function to trim wide free-text columns.
- Use the project operator to drop any uninteresting column from the result set.

You can disable result truncation by using the `notruncation` request option. We recommend that some form of limitation is still put in place.

For example:

Kusto

```
set notruncation;
MyTable | take 1000000
```

It's also possible to have more refined control over result truncation by setting the value of `truncationmaxsize` (maximum data size in bytes, defaults to 64 MB) and `truncationmaxrecords` (maximum number of records, defaults to 500,000). For example,

the following query sets result truncation to happen at either 1,105 records or 1 MB, whichever is exceeded.

Kusto

```
set truncationmaxsize=1048576;
set truncationmaxrecords=1105;
MyTable | where User=="UserId1"
```

Removing the result truncation limit means that you intend to move bulk data out of Kusto.

You can remove the result truncation limit either for export purposes by using the `.export` command or for later aggregation. If you choose later aggregation, consider aggregating by using Kusto.

Kusto provides a number of client libraries that can handle "infinitely large" results by streaming them to the caller. Use one of these libraries, and configure it to streaming mode. For example, use the .NET Framework client (Microsoft.Azure.Kusto.Data) and either set the streaming property of the connection string to *true*, or use the *ExecuteQueryV2Async()* call that always streams results. For an example of how to use *ExecuteQueryV2Async()*, see the HelloKustoV2 application.

You may also find the C# streaming ingestion sample application helpful.

Result truncation is applied by default, not just to the result stream returned to the client. It's also applied by default to any subquery that one cluster issues to another cluster in a cross-cluster query, with similar effects.

Setting multiple result truncation properties

The following apply when using `set` statements, and/or when specifying flags in client request properties.

- If `notruncation` is set, and any of `truncationmaxsize`, `truncationmaxrecords`, or `query_take_max_records` are also set - `notruncation` is ignored.
- If `truncationmaxsize`, `truncationmaxrecords` and/or `query_take_max_records` are set multiple times - the *lower* value for each property applies.

Limit on memory consumed by query operators (E_RUNAWAY_QUERY)

Kusto limits the memory that each query operator can consume to protect against "runaway" queries. This limit might be reached by some query operators, such as `join` and `summarize`, that operate by holding significant data in memory. By default the limit is 5GB (per cluster node), and it can be increased by setting the request option `maxmemoryconsumptionperiterator`:

Kusto

```
set maxmemoryconsumptionperiterator=68719476736;
MyTable | summarize count() by Use
```

When this limit is reached, a partial query failure is emitted with a message that includes the text `E_RUNAWAY_QUERY`.

text

```
The ClusterBy operator has exceeded the memory budget during evaluation.
Results may be incorrect or incomplete E_RUNAWAY_QUERY.

The DemultiplexedResultSetCache operator has exceeded the memory budget
during evaluation. Results may be incorrect or incomplete (E_RUNAWAY_QUERY).

The ExecuteAndCache operator has exceeded the memory budget during
evaluation. Results may be incorrect or incomplete (E_RUNAWAY_QUERY).

The HashJoin operator has exceeded the memory budget during evaluation.
Results may be incorrect or incomplete (E_RUNAWAY_QUERY).

The Sort operator has exceeded the memory budget during evaluation. Results
may be incorrect or incomplete (E_RUNAWAY_QUERY).

The Summarize operator has exceeded the memory budget during evaluation.
Results may be incorrect or incomplete (E_RUNAWAY_QUERY).

The TopNestedAggregator operator has exceeded the memory budget during
evaluation. Results may be incorrect or incomplete (E_RUNAWAY_QUERY).

The TopNested operator has exceeded the memory budget during evaluation.
Results may be incorrect or incomplete (E_RUNAWAY_QUERY).
```

If `maxmemoryconsumptionperiterator` is set multiple times, for example in both client request properties and using a `set` statement, the lower value applies.

An additional limit that might trigger an `E_RUNAWAY_QUERY` partial query failure is a limit on the max accumulated size of strings held by a single operator. This limit cannot be overridden by the request option above:

text

```
Runaway query (E_RUNAWAY_QUERY). Aggregation over string column exceeded the
memory budget of 8GB during evaluation.
```

When this limit is exceeded, most likely the relevant query operator is a `join`, `summarize`, or `make-series`. To work-around the limit, one should modify the query to use the shuffle query strategy. (This is also likely to improve the performance of the query.)

In all cases of `E_RUNAWAY_QUERY`, an additional option (beyond increasing the limit by setting the request option and changing the query to use a shuffle strategy) is to switch to sampling. The two queries below show how to do the sampling. The first query is a statistical sampling, using a random number generator. The second query is deterministic sampling, done by hashing some column from the data set, usually some ID.

Kusto

```
T | where rand() < 0.1 | ...

T | where hash(UserId, 10) == 1 | ...
```

Limit on memory per node

Max memory per query per node is another limit used to protect against "runaway" queries. This limit, represented by the request option `max_memory_consumption_per_query_per_node`, sets an upper bound on the amount of memory that can be used on a single node for a specific query.

Kusto

```
set max_memory_consumption_per_query_per_node=68719476736;
MyTable | ...
```

If `max_memory_consumption_per_query_per_node` is set multiple times, for example in both client request properties and using a `set` statement, the lower value applies.

If the query uses `summarize`, `join`, or `make-series` operators, you can use the shuffle query strategy to reduce memory pressure on a single machine.

Limit execution timeout

Server timeout is a service-side timeout that is applied to all requests. Timeout on running requests (queries and control commands) is enforced at multiple points in the Kusto:

- client library (if used)
- service endpoint that accepts the request
- service engine that processes the request

By default, timeout is set to four minutes for queries, and 10 minutes for control commands. This value can be increased if needed (capped at one hour).

- Various client tools support changing the timeout as part of their global or per-connection settings. For example, in Kusto.Explorer, use **Tools** > **Options*** > **Connections** > **Query Server Timeout**.
- Programmatically, SDKs support setting the timeout through the `servertimeout` property. For example, in .NET SDK this is done through a client request property, by setting a value of type `System.TimeSpan`.

Notes about timeouts

- On the client side, the timeout is applied from the request being created until the time that the response starts arriving to the client. The time it takes to read the payload back at the client isn't treated as part of the timeout. It depends on how quickly the caller pulls the data from the stream.
- Also on the client side, the actual timeout value used is slightly higher than the server timeout value requested by the user. This difference, is to allow for network latencies.
- To automatically use the maximum allowed request timeout, set the client request property `norequesttimeout` to `true`.

> ⓘ **Note**
>
> See **set timeout limits** for a step-by-step guide on how to set timeouts in the Azure Data Explorer web UI, Kusto.Explorer, Kusto.Cli, Power BI, and when using an SDK.

Limit on query CPU resource usage

Kusto lets you run queries and use as much CPU resources as the cluster has. It attempts to do a fair round-robin between queries if more than one is running. This method

yields the best performance for query-defined functions. At other times, you may want to limit the CPU resources used for a particular query. If you run a "background job", for example, the system might tolerate higher latencies to give concurrent inline queries high priority.

Kusto supports specifying two client request properties when running a query. The properties are *query_fanout_threads_percent* and *query_fanout_nodes_percent*. Both properties are integers that default to the maximum value (100), but may be reduced for a specific query to some other value.

The first, *query_fanout_threads_percent*, controls the fanout factor for thread use. When this property is set 100%, the cluster will assign all CPUs on each node. For example, 16 CPUs on a cluster deployed on Azure D14 nodes. When this property is set to 50%, then half of the CPUs will be used, and so on. The numbers are rounded up to a whole CPU, so it's safe to set the property value to 0.

The second, *query_fanout_nodes_percent*, controls how many of the query nodes in the cluster to use per subquery distribution operation. It functions in a similar manner.

If `query_fanout_nodes_percent` or `query_fanout_threads_percent` are set multiple times, for example, in both client request properties and using a `set` statement - the lower value for each property applies.

Limit on query complexity

During query execution, the query text is transformed into a tree of relational operators representing the query. If the tree depth exceeds an internal threshold, the query is considered too complex for processing, and will fail with an error code. The failure indicates that the relational operators tree exceeds its limits.

The following examples show common query patterns that can cause the query to exceed this limit and fail:

- a long list of binary operators that are chained together. For example:

Kusto

```
T
| where Column == "value1" or
        Column == "value2" or
        .... or
        Column == "valueN"
```

For this specific case, rewrite the query using the in() operator.

Kusto

```
T
| where Column in ("value1", "value2".... "valueN")
```

- a query which has a union operator that is running too wide schema analysis especially that the default flavor of union is to return "outer" union schema (meaning – that output will include all columns of the underlying table).

The suggestion in this case is to review the query and reduce the columns being used by the query.

Next steps

- Optimize for high concurrency with Azure Data Explorer
- Query best practices

Feedback

Partial query failures

Article • 03/07/2022

A *partial query failure* is a failure to run the query that gets detected only after the query has started the actual execution phase. By that time, Kusto has already returned the HTTP status line `200 OK` back to the client, and cannot "take it back", so it has to indicate the failure in the result stream that carries the query results back to the client. (In fact, it may have already returned some result data back to the caller.)

There are several kinds of partial query failures:

- Runaway queries: Queries that take up too many resources.
- Result truncation: Queries whose result set has been truncated as it exceeded some limit.
- Overflows: Queries that trigger an overflow error.

Partial query failures can be reported back to the client in one of two ways:

- As part of the result data (a special kind of record indicates that this is not the data itself). This is the default way.
- As part of the "QueryStatus" table in the result stream. This is done by using the `deferpartialqueryfailures` option in the request's `properties` slot (`Kusto.Data.Common.ClientRequestProperties.OptionDeferPartialQueryFailures`). Clients that do that take on the responsibility to consume the entire result stream from the service, locate the `QueryStatus` result, and make sure no record in this result has a `Severity` of `2` or smaller.

Feedback

Query result set has exceeded the internal ... limit

Article • 03/07/2022

A *query result set has exceeded the internal ... limit* is a kind of partial query failure that happens when the query's result has exceeded one of two limits:

- A limit on the number of records (`record count limit`, set by default to 500,000)
- A limit on the total amount of data (`data size limit`, set by default to 67,108,864 (64MB))

There are several possible courses of action:

- Change the query to consume fewer resources. For example, you can:
  - Limit the number of records returned by the query using the take operator or adding additional where clauses
  - Try to reduce the number of columns returned by the query. Use the project operator, the project-away operator, or the project-keep operator
  - Use the summarize operator to get aggregated data
- Increase the relevant query limit temporarily for that query. For more information, see **Result truncation** under query limits)

> **Note**
>
> We don't recommend that you increase the query limit, since the limits exist to protect the cluster. The limits make sure that a single query doesn't disrupt concurrent queries running on the cluster.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Runaway queries

Article • 03/07/2022

A *runaway query* is a kind of partial query failure that happens when some internal query limit was exceeded during query execution.

For example, the following error may be reported: `HashJoin operator has exceeded the memory budget during evaluation. Results may be incorrect or incomplete.`

There are several possible courses of action.

- Change the query to consume fewer resources. For example, if the error indicates that the query result set is too large, you can:
  - Limit the number of records returned by the query by
    - Using the take operator
    - Adding additional where clauses
  - Reduce the number of columns returned by the query by
    - Using the project operator
    - Using the project-away operator
    - Using the project-keep operator
  - Use the summarize operator to get aggregated data.
- Increase the relevant query limit temporarily for that query. For more information, see query limits - limit on memory per iterator. This method, however, isn't recommended. The limits exist to protect the cluster and to make sure that a single query doesn't disrupt concurrent queries running on the cluster.

Feedback

Overflows

Article • 03/07/2022

An overflow occurs when the result of a computation is too large for the destination type. The overflow usually leads to a partial query failure.

For example, the following query will result in an overflow.

Kusto

```
let Weight = 92233720368547758;
range x from 1 to 3 step 1
| summarize percentilesw(x, Weight * 100, 50)
```

Kusto's `percentilesw()` implementation accumulates the `Weight` expression for values that are "close enough". In this case, the accumulation triggers an overflow because it doesn't fit into a signed 64-bit integer.

Usually, overflows are a result of a "bug" in the query, since Kusto uses 64-bit types for arithmetic computations. The best course of action is to look at the error message, and identify the function or aggregation that triggered the overflow. Make sure the input arguments evaluate to values that make sense.

Feedback