Query best practices

Article • 06/29/2023

Here are several best practices to follow to make your query run faster.

In short

Action	Use	Don't use	Notes
Reduce the amount of data being queried	Use mechanisms such as the `where` operator to reduce the amount of data being processed.		See below for efficient ways to reduce the amount of data being processed.
Avoid using redundant qualified references	When referencing local entities, use the unqualified name.		See below for more on the subject.
`datetime` **columns**	Use the `datetime` data type.	Don't use the `long` data type.	In queries, don't use unix time conversion functions, such as `unixtime_milliseconds_todatetime()`. Instead, use update policies to convert unix time to the `datetime` data type during ingestion.
String operators	Use the `has` operator	Don't use `contains`	When looking for full tokens, `has` works better, since it doesn't look for substrings.
Case-sensitive operators	Use `==`	Don't use `=~`	Use case-sensitive operators when possible.
	Use `in`	Don't use `in~`	
	Use `contains_cs`	Don't use `contains`	If you can use `has`/`has_cs` and not use `contains`/`contains_cs`, that's even better.
Searching text	Look in a specific column	Don't use `*`	`*` does a full text search across all columns.

Action	Use	Don't use	Notes
Extract fields from dynamic objects across millions of rows	Materialize your column at ingestion time if most of your queries extract fields from dynamic objects across millions of rows.		This way, you'll only pay once for column extraction.
Lookup for rare keys/values in dynamic objects	Use `MyTable | where DynamicColumn has "Rare value" | where DynamicColumn.SomeKey == "Rare value"`	Don't use `MyTable | where DynamicColumn.SomeKey == "Rare value"`	This way, you filter out most records, and do JSON parsing only of the rest.
`let` statement with a value that you use more than once	Use the materialize() function		For more information on how to use `materialize()`, see materialize(). For more information, see Optimize queries that use named expressions.
Apply conversions on more than 1 billion records	Reshape your query to reduce the amount of data fed into the conversion.	Don't convert large amounts of data if it can be avoided.	
New queries	Use `limit [small number]` or `count` at the end.		Running unbound queries over unknown data sets may yield GBs of results to be returned to the client, resulting in a slow response and a busy cluster.
Case-insensitive comparisons	Use `Col =~ "lowercasestring"`	Don't use `tolower(Col) == "lowercasestring"`	
Compare data already in lowercase (or uppercase)	`Col == "lowercasestring"` (or `Col == "UPPERCASESTRING"`)	Avoid using case insensitive comparisons.	
Filtering on columns	Filter on a table column.	Don't filter on a calculated column.	

Action	Use	Don't use	Notes
	Use `T | where predicate(*Expression*)`	Don't use `T | extend _value = *Expression* | where predicate(_value)`	
summarize operator	Use the hint.shufflekey=<key> when the `group by keys` of the summarize operator are with high cardinality.		High cardinality is ideally above 1 million.
join operator	Select the table with the fewer rows to be the first one (left-most in query).		
	Use `in` instead of left semi `join` for filtering by a single column.		
Join across clusters	Across clusters, run the query on the "right" side of the join, where most of the data is located.		
Join when left side is small and right side is large	Use hint.strategy=broadcast		Small refers to up to 100MB of data.
Join when right side is small and left side is large	Use the lookup operator instead of the `join` operator		If the right side of the lookup is larger than several tens of MBs, the query will fail.
Join when both sides are too large	Use hint.shufflekey=<key>		Use when the join key has high cardinality.
Extract values on column with strings sharing the same format or pattern	Use the parse operator	Don't use several `extract()` statements.	For example, values like `"Time = <time>, ResourceId = <resourceId>, Duration = <duration>, ...."`

Action	Use	Don't use	Notes
extract() function	Use when parsed strings don't all follow the same format or pattern.		Extract the required values by using a REGEX.
materialize() function	Push all possible operators that will reduce the materialized data set and still keep the semantics of the query.		For example, filters, or project only required columns. For more information, see Optimize queries that use named expressions.
Use materialized views	Use materialized views for storing commonly used aggregations. Prefer using the `materialized_view()` function to query materialized part only		`materialized_view('MV')`

Reduce the amount of data being processed

A query's performance depends directly on the amount of data it needs to process. The less data is processed, the quicker the query (and the fewer resources it consumes). Therefore, the most important best-practice is to structure the query in such a way that reduces the amount of data being processed.

> ⓘ Note
>
> In the discussion below, it is important to have in mind the concept of **filter selectivity**. Selectivity is what percentage of the records get filtered-out when filtering by some predicate. A highly-selective predicate means that only a handful of records remain after applying the predicate, reducing the amount of data that needs to then be processed effectively.

In order of importance:

- Only reference tables whose data is needed by the query. For example, when using the `union` operator with wildcard table references, it is better from a performance point-of-view to only reference a handful of tables, instead of using a wildcard (`*`) to reference all tables and then filter data out using a predicate on the source table name.

- Take advantage of a table's data scope if the query is relevant only for a specific scope. The table() function provides an efficient way to eliminate data by scoping it according to the caching policy (the *DataScope* parameter).
- Apply the `where` query operator immediately following table references.
- When using the `where` query operator, a judicious use of the order of predicates (in a single operator, or with a number of consecutive operators, it doesn't matter which) can have a significant effect on the query performance, as explained below.
- Apply whole-shard predicates first. This means that predicates that use the extent_id() function should be applied first, as are predicates that use the extent_tags() function and predicates that are very selective over the table's data partitions (if defined).
- Then apply predicates that act upon `datetime` table columns. Kusto includes a very efficient index on such columns, often eliminating whole data shards completely without needing to access those shards.
- Then apply predicates that act upon `string` and `dynamic` columns, especially such predicates that apply at the term-level. The predicates should be ordered by the selectivity (for example, searching for a user ID when there are millions of users is very selective and usually is a term search for which the index is very efficient.)
- Then apply predicates that are selective and are based on numeric columns.
- Last, for queries that scan a table column's data (for example, for predicates such as `contains "@!@!" that have no terms and don't benefit from indexing), order the predicates such that the ones that scan columns with less data will be first. This reduces the need to decompress and scan large columns.

Avoid using redundant qualified references

Entities such as tables and materialized views are referenced by name. For example, the table `T` can be referenced as simply `T` (the *unqualified* name), or by using a database qualifier (e.g. `database("DB").T` when the table is in a database called `DB`), or by using a fully-qualified name (e.g. `cluster("X.Y.kusto.windows.net").database("DB").T`).

It is a best practice to avoid using name qualifications when they are redundant, for the following reasons:

1. Unqualified names are easier to identify (for a human reader) as belonging to the database-in-scope.

2. Referencing database-in-scope entities is always at least as fast, and in some cases much faster, then entities that belong to other databases (especially when those databases are in a different cluster.) Avoiding qualified names helps the reader to do the right thing.

> **Note**
>
> This is not to say that qualified names are bad for performance. In fact, Kusto is able in most cases to identify when a fully qualified name references an entity belonging to the database-in-scope and "short-circuit" the query so that it is not regarded as a cross-cluster query. However, we do recommend to not rely on this when not necessary, for the reasons specified above.

Feedback

Optimize queries that use named expressions

Article • 06/06/2023

This article discusses how to optimize repeat use of named expressions in a query.

In Kusto Query Language, you can bind names to complex expressions in several different ways:

- In a let statement
- In the as operator
- In the formal parameters list of user-defined functions

When you reference these named expressions in a query, the following steps occur:

1. The calculation within the named expression is evaluated. This calculation produces either a scalar or tabular value.
2. The named expression is replaced with the calculated value.

If the same bound name is used multiple times, then the underlying calculation will be repeated multiple times. When is this a concern?

- When the calculations consume many resources and are used many times.
- When the calculation is non-deterministic, but the query assumes all invocations to return the same value.

Mitigation

To mitigate these concerns, you can materialize the calculation results in memory during the query. Depending on the way the named calculation is defined, you'll use different materialization strategies:

Tabular functions

Use the following strategies for tabular functions:

- **let statements and function parameters**: Use the materialize() function.
- **as operator**: Set the `hint.materialized` hint value to `true`.

For example, the following query uses the non-deterministic tabular sample operator:

> ⓘ Note
>
> Tables aren't sorted in general, so any table reference in a query is, by definition, non-deterministic.

Behavior without using the materialize function

Run the query

```
Kusto
range x from 1 to 100 step 1
| sample 1
| as T
| union T
```

Output

x
63
92

Behavior using the materialize function

Run the query

```
Kusto
range x from 1 to 100 step 1
| sample 1
| as hint.materialized=true T
| union T
```

Output

x
95
95

Scalar functions

Non-deterministic scalar functions can be forced to calculate exactly once by using toscalar().

For example, the following query uses the non-deterministic function, rand():

Run the query

Kusto

```
let x = () {rand(1000)};
let y = () {toscalar(rand(1000))};
print x, x, y, y
```

Output

print_0	print_1	print_2	print_3
166	137	70	70

See also

- Let statement
- as operator
- toscalar()

Feedback