MariaDB optimizer For “11.0”

MariaDB Server Fest November 2022

Michael Widenius
CTO @ MariaDB

Talk overview

- This talk is about the optimizer features coming in the next main MariaDB release, probably called 11.0.
- Why these changes were done
- I try to describe all “user visible” changes in the “improved MariaDB optimizer”
  - Replace several **rule** based choices with **cost** based
    - (A big part of the old optimizer was cost based, but not all...)
  - Major cost changes
    - Costs are now based on timing of engine sub operations and **expressed in microseconds** instead of ‘key lookup’, ‘row lookup’ or ‘access a disk block’.
  - User changeable optimizer costs
  - Fixes in selectivity (chance of a row passing the WHERE clause) calculations.
  - Optimizer trace changes
  - When / why these changes matter to you

MariaDB

Background for the optimizer changes

- It started with MDEV-26974 “Improve selectivity and related costs in optimizer” in September 2021
- The intention was to fix a selectivity issue where the optimizer could sometimes calculate selectivity to be > 1.0
- While fixing this I discovered that there were a lot of other things that were “far from optimal” that should be fixed.
- In August 2022, I concluded that the old costs were not good enough to be able to calculate the best plan
  - Changed costs to be in milliseconds and created a program, check_costs.pl, to calculate and verify costs.
- Now, a bit more than one year later, most of the ‘critically needed issues’ are fixed…
  - The selectivity tree has more than 60 commits (a few very big ones).

The intention of the optimizer changes

- Be able to find the '**best table combination and best access plan**' for a query.
  - The original optimizer cost model was not very good if there were no good indexes.
    - Replace most of the remaining rule based choices with cost based choices.
    - Take into account that different engines have different characteristics (Memory vs InnoDB)
- Allow the user to **fine tune the optimizer costs** for their environment.
  - (Hopefully they never need to do that, but it is now possible)
- Easier to compare query costs and also quickly see 'if a cost is reasonable'.
  - Having **costs in microseconds** helps to verify if a cost is 'totally wrong'
  - Optimizer trace writes out a lot more information about the costs!

The intention for the future is to be able to enable all optimizer_switch options by default! For this we needed a better cost model, like the new one, as a base!

MariaDB

Optimizer trace added to MariaDB 10.4

The new optimizer trace has made it possible to start improving the optimizer.
To use it one should do:

set optimizer_trace="enabled=on";
SELECT ...
select * from information_schema.OPTIMIZER_TRACE;

In an mtr test (mariadb-test-run) one can (starting with 10.5) use **--optimizer_trace** before a **SELECT or EXPLAIN** query that produces wrong results, to find out what is different from before.

optimizer_trace has enabled me to start working on the biggest change in optimizer cost calculation since MariaDB 10.3/10.5!

11.0 has much more information in the optimizer trace!

MariaDB

Selectivity (bug fix)

- Starting from MariaDB 10.4.1 has **optimizer_use_condition_selectivity=4**
- In some cases the selectivity calculation is wrong (selectivity becomes > 1) and one gets a bad plan.
- Current workaround is to use **optimizer_use_condition_selectivity=1** if a plan is bad.

- Selectivity calculations are now fixed. There are now asserts in place in all selectivity calculations that ensure this cannot happen again.
- **The optimizer now uses the most optimistic (smallest number of rows) access method when estimating rows count.**
  - One effect of this is that ‘explain extended’ now has a more accurate number for “filtered”.

Derived tables and union can now create distinct key

- Temporary derived tables are now creating unique keys to speed up searches.

Here is a diff from the commit: now eq_ref (unique key lookup) instead of ref

```
EXPLAIN UPDATE        t1, t2 SET a = 10 WHERE a IN (SELECT * FROM (SELECT b FROM t2 ORDER BY b
LIMIT 2,2) x);
 id      select_type      table    type     possible_keys    key      key_len ref      rows     Extra
 1       PRIMARY t1       ALL      NULL     NULL     NULL    NULL     3       Using where
-1       PRIMARY <derived3>       ref      key0     key0    5        test.t1.a       2    FirstMatch(t1)
+1       PRIMARY <derived3>       eq_ref   distinct_key     distinct_key     5        test.t1.a        1

 1       PRIMARY t2       ALL      NULL     NULL     NULL    NULL     3
 3       DERIVED t2       ALL      NULL     NULL     NULL    NULL     3       Using filesort
```

MariaDB

New cost calculations

- Cost calculations for filesort, Unique, filters, join_cache, materialization are updated.
  - The consequences for these are:
    - MariaDB is more likely to use an index for order by
    - MariaDB will use filters a bit more than before.
    - Materialization costs are now a bit higher
- Cost of “Using index for group-by” corrected.
  - MariaDB will use “index for group by” optimization more optimal now.
- The disk access cost is now assuming SSD!
- When counting disk accesses, we assume that **all read rows are cached** for the duration of the query. If this calculation would not be done, the cost of joining a big table with a small one would be unreasonable high!

The new storage engine costs

- Cost calculations changed from using ‘disk/row/index’ access to **microseconds**.
  - As part of this, the base costs (table_scan, index_scan, key_look, row_lookup) have been split into smaller parts:

select * from information_schema.optimizer_costs where engine="innodb"

```
       OPTIMIZER_DISK_READ_COST: 10.240000
OPTIMIZER_INDEX_BLOCK_COPY_COST: 0.035600
     OPTIMIZER_KEY_COMPARE_COST: 0.011361
        OPTIMIZER_KEY_COPY_COST: 0.015685
      OPTIMIZER_KEY_LOOKUP_COST: 0.791120
   OPTIMIZER_KEY_NEXT_FIND_COST: 0.099000
      OPTIMIZER_DISK_READ_RATIO: 0.020000
        OPTIMIZER_ROW_COPY_COST: 0.060870
      OPTIMIZER_ROW_LOOKUP_COST: 0.765970
   OPTIMIZER_ROW_NEXT_FIND_COST: 0.070130
   OPTIMIZER_ROWID_COMPARE_COST: 0.002653
      OPTIMIZER_ROWID_COPY_COST: 0.002653
```

Note that the above costs are in microseconds, while the query costs (in optimizer_trace) is in milliseconds!
Docs/optimizer_costs.txt explains in detail how the costs are calculated!

MariaDB

The new (important) SQL level costs

show variables like "optimizer%cost";

```
+------------------------------+-----------+
| Variable_name                | Value     |
+------------------------------+-----------+
| optimizer_disk_read_cost     | 10.240000 |
...
| optimizer_where_cost         | 0.032000  |
+------------------------------+-----------+
```

show variables like "optimizer%ratio"

```
+--------------------------+----------+
| Variable_name            | Value    |
+--------------------------+----------+
| optimizer_disk_read_ratio | 0.020000 |
+--------------------------+----------+
```

MariaDB

Verifying the optimizer costs (All data is in memory for this test)

check_costs.pl --engine=aria

```
…
Timing table access for query: table scan
select sum(l_quantity) from test.check_costs_aria
explain:
1  SIMPLE  check_costs_aria  ALL  1000000
table_scan  time: 108.057814 ms  cost-where: 107.3483   cost: 139.3483
…
Cost/time ratio for different scans types
table scan            cost:  107.3483  time:  108.0578  cost/time:    0.9934
index scan            cost:   98.1252  time:   88.2091  cost/time:    1.1124
range scan            cost:  309.7452  time:  337.0256  cost/time:    0.9191
eq_ref_index_join     cost:  499.5453  time:  495.4494  cost/time:    1.0083
eq_ref_cluster_join   cost:  499.5453  time:  497.5671  cost/time:    1.0040
eq_ref_join           cost:  711.1653  time:  762.0677  cost/time:    0.9332
eq_ref_btree          cost:  711.1653  time:  760.3875  cost/time:    0.9353
```

MariaDB

Example of key read cost calculation

`IO_AND_CPU_COST handler::keyread_time(index, ranges, rows)`

Calculates the number of disk blocks that we expect to access when reading trough one index, with a given set of ranges a given set of rows.
It also multiplies the number of blocks with INDEX_BLOCK_COPY_COST.

The full cost of the index read is then calculated in ha_keyread_time():

```
keyread_time() +
  ranges * KEY_LOOKUP_COST +
  (rows - ranges) * KEY_NEXT_FIND_COST;
```

For accepted rows when doing an index read, we have ha_keyread_and_compare_time() which does:

```
ha_keyread_time() + rows * (KEY_COPY_COST + WHERE_COST)
```

Example of row read cost calculation

If we are doing reading a row trough a key, we have to use the cost of reading the row:

ha_keyread_time() + ha_rnd_pos_time()

Here is the actual code:

```
virtual IO_AND_CPU_COST rnd_pos_time(ha_rows rows)
{
  double r= rows2double(rows);
  return
  {
    r * ((stats.block_size + IO_SIZE -1 )/IO_SIZE),    // Blocks read
    r * INDEX_BLOCK_COPY_COST                          // Copy block from cache
  };
}
inline IO_AND_CPU_COST ha_rnd_pos_time(ha_rows rows)
{
  IO_AND_CPU_COST cost= rnd_pos_time(rows);
  set_if_smaller(cost.io, (double) row_blocks()); // Limit the blocks to the number of blocks in the file
  cost.cpu+= rows2double(rows) * (ROW_LOOKUP_COST + ROW_COPY_COST);
  return cost;
}
```

MariaDB

Rule based -> Cost based

- The decision to use an index (and which index) for resolving ORDER BY/GROUP BY where only partly cost based before.
- The old optimizer would limit the number of ‘expected key lookups’ to 10% of the number of rows. This would cause the optimizer to use an index to scan a big part of a table when a full table scan would be much faster.
  This code is now removed.
- InnoDB would limit the number of rows in a range to 50% of the total rows, which would confuse the optimizer for big ranges. The cap is now removed.
- If there was a usable filter for an index, it was sometimes used without checking the complete cost of the filter.
- ‘Aggregate distinct optimization with indexes’ is now cost based.
  - “Using index for group-by (scanning)” → “Using index for group-by”

Other things

- A lot of small changes to improve performance
  - Changed some critical functions to be inline
  - Improved rowid_filter filling code
  - More caching of values
  - Simplified code (removed extra calls that were not needed)
- Many (!) more code comments to existing code.
- Some old MariadB bugs in Jira were solved by the new code.
- Some small improvements to **LIMIT**
- Indexes can now be used for ORDER BY/GROUP BY in sub queries (instead of filesort)
- Aria tables now supports rowid_filtering

Some other plan changes

- We now prefer indexes with more used key parts if the number of resulting rows is the same.
  - Where key_part_1 = 1 **and** key_part_2 < 10

- For very small tables, index lookup is preferred over table scan
  - This is mainly because of the mysql-run-test (mtr) test suite which has mostly small tables.
  - This can be changed by setting OPTIMIZER_SCAN_SETUP_COST=0
  - Normally this should not matter for end users.

- Do not report in EXPLAIN scans on clustered primary keys as ‘Using index’.
  - This is not an index scan, it is a table scan!
  - Maybe we should instead report ‘Using clustered index’ ?

The most important new optimizer cost variables

optimizer_disk_read_ratio	0.020000	The chance that an engine-block is cached
optimizer_disk_read_cost	10.240000	Time to read a 4K block from an SSD
optimizer_where_cost	0.032000	Time to execute the WHERE clause This time is added to all 'accepted' rows
optimizer_scan_setup_cost	10.000000	Cost added to all full table or index scans

The above variables *(in microseconds)* will ensure that if tuning is needed for the new cost calculations, one should be able to fix it by just adjusting one of the above variables in the MariaDB config file.

For example, increasing **optimizer_where_cost** will cause the optimizer to choose plans with less estimated rows.

Changing cost variables

All engine and “sql level” cost variables can be changed via mariadbd startup options, in config files or dynamically using SQL.

```
set session optimizer_where_cost=1.0;
set global innodb.OPTIMIZER_DISK_READ_COST=100;
```

- The “`default`” engine contains the default costs for all storage engines.
  - When a new engine is loaded, the default costs are taken from the “default” engine and then the engine updates its own internal costs and adds the user configured costs.
- To keep things fast, engine specific costs are stored in the table definition (TABLE_SHARE). This means that if one changes the cost for an engine, it will only take effect when new, not previously cached tables are accessed.
- You can use `flush tables` to force the table to use the new costs at next access.

When optimizer cost calculation changes, unexpected things can happen to existing applications

- The new optimizer should be able to do a better choice when to use **table scan, index scan, index_merge, hash** and other join methods needed when key lookup cannot be used.
- Most applications, which are properly using keys, should be unaffected.
  - Simple queries will work as before
  - Most complex queries (with many tables) should perform equal *or better* than before.
- The new optimizer costs may need future tuning to be ‘perfect’ for most. If needed, this will be done over a few MariaDB releases.
- As most tuning can be done by just adjusting the cost variables, this can be done quickly with no code changes even by users.

When does the optimizer changes matter to you

- The new optimizer should be able to find a better plan
  - If you are using queries with more than two tables
  - If you have indexes with a lot of identical values
  - If you are using ranges that cover more than 10% of a table
    - … WHERE key between 1 and 1000 -- Table has values 1-2000
  - If you have complex queries when not all used columns are or can be indexed
    - In which case you may need to depend on selectivity to get the right plan
  - If you are using queries mixing different storage engines
    - Like using both InnoDB and Memory engine in the same query.
  - If you have had to use FORCE INDEX to get a good plan.
  - If using ANALYZE TABLE made your plans worse (or not good enough)
  - If your queries have lots of derived tables (subselects)
  - Using ORDER BY / GROUP BY that could be resolved via indexes

State of things

- All changes (except minor tuning based on input from user or performance testing) are already done!
- The code can be found in the bb-10.11-selectivity-nov branch
- It will probably to be renamed next week to 11.0-alpha and changed to -beta by the end of the month.
- The optimizer will probably be the only major change in this tree!
- We want the release to become ‘stable’ quickly and many testers would help!
- We encourage everyone to test this and give us feedback (through Jira) so that we can fix any bugs ASAP! Please consider putting 11.0-beta as an extra slave to your existing production and give us FEEDBACK!

A last note: Going through the optimizer code has given me a lot of ideas for cleanups that could be done. This will be done in next major MariaDB releases.

MariaDB

Acknowledgments

- A big thanks to Sergei Petrunia for optimizer_trace and for reviews and always been available to explain some of the optimizer internals to me!
- Thanks to Vicențiu Ciorbaru for adding cost calculations for filesort!
- Thanks to Andrew Hutchings for helping on the ColumnStore front.

MariaDB

Thank you

Any questions?