MariaDB Fest
September 2020

ANALYZE for statements

MariaDB's hidden gem

Includes a comparison with MySQL's EXPLAIN ANALYZE

Sergei Petrunia
MariaDB

ANALYZE for statements

- Introduced in MariaDB 10.1 (Oct, 2015)
  - Got less recognition than it deserves
- Was improved in the next MariaDB versions
  - Based on experience
- MySQL 8.0.18 (Oct, 2019) introduced EXPLAIN ANALYZE
  - Very similar feature, let’s compare.

The plan

- ANALYZE In MariaDB
  - ANALYZE
  - ANALYZE FORMAT=JSON
- EXPLAIN ANALYZE in MySQL
  - Description and comparison

Background: EXPLAIN

```
explain select *
from lineitem, orders
where o_orderkey=l_orderkey and
      o_orderdate between '1990-01-01' and '1998-12-06' and
      l_extendedprice > 1000000
```

```
+--+----------+--------+----+------------+-------+-------+----------------+-------+--------+-----------+
|id|select_type|table   |type|possible_keys|key    |key_len|ref             |rows   |filtered|Extra      |
+--+----------+--------+----+------------+-------+-------+----------------+-------+--------+-----------+
|1 |SIMPLE     |orders  |ALL |PRIMARY,i_...|NULL   |NULL   |NULL            |1504278|  50.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4      |orders.o_orderkey|2      | 100.00 |Using where|
+--+----------+--------+----+------------+-------+-------+----------------+-------+--------+-----------+
```

- Sometimes problem is apparent
- Sometimes not
  - Query plan vs reality?
  - Where the time was spent?

ANALYZE vs EXPLAIN

EXPLAIN

- Optimize the query
- Produce EXPLAIN output

ANALYZE

- Optimize the query
- Run the query
  - Collect execution statistics
  - Discard query output
- Produce EXPLAIN output
  - With also execution statistics

Tabular ANALYZE statement

```
analyze select *
from lineitem, orders
where o_orderkey=l_orderkey and
      o_orderdate between '1990-01-01' and '1998-12-06' and
      l_extendedprice > 1000000
```

```
+--+-----------+-------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|id|select_type|table  |type|possible_keys|key    |key_len|ref              |rows   |r_rows |filtered|r_filtered|Extra      |
+--+-----------+-------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|1 |SIMPLE     |orders |ALL |PRIMARY,i_...|NULL   |NULL   |NULL             |1504278|1500000|  50.00 |   100.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4      |orders.o_orderkey|2      |4.00   | 100.00 |     0.00 |Using where|
+--+-----------+-------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
```

r_ is for “real”

- **r_rows –** observed #rows
- **r_filtered** – observed condition selectivity.

Interpreting **r_rows**

```
+--+-----------+-------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
|id|select_type|table  |type|possible_keys|key   |key_len|ref              |rows   |r_rows |filtered|r_filtered|Extra      |
+--+-----------+-------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
|1 |SIMPLE     |orders |ALL |PRIMARY,i_...|NULL  |NULL   |NULL             |1504278|1500000|  50.00 |   100.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4     |orders.o_orderkey|2      |4.00   | 100.00 |     0.00 |Using where|
+--+-----------+-------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
```

- ALL/index
  - r_rows ≈ rows
  - Different in case of LIMIT or subqueries
- range/index_merge
  - Up to ~2x difference with InnoDB
  - Bigger difference in edge cases (IGNORE INDEX?)

Interpreting **r_rows** (2)

```
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|id|select_type|table   |type|possible_keys|key    |key_len|ref              |rows   |r_rows |filtered|r_filtered|Extra      |
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|1 |SIMPLE     |orders  |ALL |PRIMARY,i_...|NULL   |NULL   |NULL             |1504278|1500000|  50.00 |   100.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4      |orders.o_orderkey|2      |4.00   | 100.00 |     0.00 |Using where|
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
```

For **ref** access

- rows is AVG(records for a key)
- Some discrepancy is normal
- Big discrepancy (>10x) is worth investigating
  - rows=1, r_rows >> rows ? No index statistics (ANALYZE TABLE)
  - Column has lots of NULL values? (innodb_stats_method)
  - Skewed data distribution?
    - Complex, IGNORE INDEX.

Interpreting **r_filtered**

```
analyze select *
from lineitem, orders
where o_orderkey=l_orderkey and
      o_orderdate between '1990-01-01' and '1998-12-06' and
      l_extendedprice > 1000000
```

```
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|id|select_type|table   |type|possible_keys|key    |key_len|ref              |rows   |r_rows |filtered|r_filtered|Extra      |
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
|1 |SIMPLE     |orders  |ALL |PRIMARY,i_...|NULL   |NULL   |NULL             |1504278|1500000|  50.00 |   100.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4      |orders.o_orderkey|2      |4.00   | 100.00 |     0.00 |Using where|
+--+-----------+--------+----+-------------+-------+-------+-----------------+-------+-------+--------+----------+-----------+
```

- **(r_)filtered** is selectivity of “Using where”
- **r_filtered** << 100% → reading and discarding lots of rows
  - Check if conditions allow index use
    - Use EXPLAIN|ANALYZE FORMAT=JSON to see the condition
  - Consider adding indexes
  - Don't chase r_filtered=100.00, tradeoff between reads and writes.

r_filtered and query plans

- **filtered**
  - Shows how many rows will be removed from consideration
  - Is important for N-way join optimization
- **r_filtered ≠ filtered**
  - Optimizer doesn't know condition selectivity → poor plans
  - Consider collecting histogram(s) on columns used by the condition

Tabular **ANALYZE** summary

```
+--+-----------+--------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
|id|select_type|table   |type|possible_keys|key   |key_len|ref              |rows   |r_rows |filtered|r_filtered|Extra      |
+--+-----------+--------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
|1 |SIMPLE     |orders  |ALL |PRIMARY,i_...|NULL  |NULL   |NULL             |1504278|1500000|  50.00 |   100.00 |Using where|
|1 |SIMPLE     |lineitem|ref |PRIMARY,i_...|PRIMARY|4     |orders.o_orderkey|2      |4.00   | 100.00 |     0.00 |Using where|
+--+-----------+--------+----+-------------+------+-------+-----------------+-------+-------+--------+----------+-----------+
```

- New columns:
  - **r_rows**
  - **r_filtered**
- Can check estimates vs reality

ANALYZE FORMAT=JSON

EXPLAIN FORMAT=JSON + ANALYZE = ANALYZE FORMAT=JSON

Basics

```
analyze format=json
select count(*) from orders where o_totalprice > 50000
```

```
{
  "query_block": {
    "select_id": 1,
    "r_loops": 1,
    "r_total_time_ms": 190.15729,
    "table": {
      "table_name": "orders",
      "access_type": "ALL",
      "r_loops": 1,
      "rows": 1492405,
      "r_rows": 1500000,
      "r_table_time_ms": 142.0726678,
      "r_other_time_ms": 48.07306875,
      "filtered": 100,
      "r_filtered": 85.9236,
      "attached_condition": "orders.o_totalprice > 50000"
    }
  }
}
```

- **r_rows**
- **r_filtered**

- **r_loops** – number of times the operation was executed
- **r_..._time_ms** – total time spent
  - **r_table_time_ms** – time spent reading data from the table
  - **r_other_time_ms** – time spent checking the WHERE, making lookup key, etc

A complex query: join

```
analyze format=json
select *
from
  lineitem, orders
where
  o_orderkey=l_orderkey and
  o_orderdate between '1990-01-01'
                  and '1998-12-06' and
  l_extendedprice > 1000000
```

- For each table, check
  - **r_table_time_ms,**

    r_other_time_ms
  - **r_loops**
- Instantly shows the problem areas


```
{
  "query_block": {
    "select_id": 1,
    "r_loops": 1,
    "r_total_time_ms": 4056.659499,
    "table": {
      "table_name": "orders",
      "access_type": "ALL",
      "possible_keys": ["PRIMARY", "i_o_orderdate"],
      "r_loops": 1,
      "rows": 1492405,
      "r_rows": 1500000,
      "r_table_time_ms": 323.3849353,
      "r_other_time_ms": 118.576661,
      "filtered": 49.99996567,
      "r_filtered": 100,
      "attached_condition": "orders.o_orderDATE between '1990-
    },
    "table": {
      "table_name": "lineitem",
      "access_type": "ref",
      "possible_keys": ["i_l_orderkey", "i_l_orderkey_quantity"],
      "key": "PRIMARY",
      "key_length": "4",
      "used_key_parts": ["l_orderkey"],
      "ref": ["dbt3sf1.orders.o_orderkey"],
      "r_loops": 1500000,
      "rows": 2,
      "r_rows": 4.00081,
      "r_table_time_ms": 3454.758327,
      "r_other_time_ms": 159.9274175,
      "filtered": 100,
      "r_filtered": 0,
      "attached_condition": "lineitem.l_extendedprice > 1000000"
    }
  }
}
```

A complex query: subquery

```
analyze format=json
select *
from orders OT
where
  o_orderdate between '1998-06-01' and
                      '1998-12-06' and
  o_totalprice >
   0.9 * (select
            sum(o_totalprice)
          from
            orders
          where
            o_custkey=OT.o_custkey)
```

- Each subquery is a **query_block**
- It has **r_total_time_ms, r_loops**
  - time includes children' time
- Narrowing down problems in complex queries is now easy!

- Can also see subquery cache
  - was used but had r_rit_ratio=0 so it disabled itself.

```
{
  "query_block": {
    "select_id": 1,
    "r_loops": 1,
    "r_total_time_ms": 1104.099893,
    "table": {
      "table_name": "OT",
      "access_type": "range",
      "possible_keys": ["i_o_orderdate"],
      "key": "i_o_orderdate",
      "key_length": "4",
      "used_key_parts": ["o_orderDATE"],
      "r_loops": 1,
      "rows": 78708,
      "r_rows": 39162,
      "r_table_time_ms": 63.71686522,
      "r_other_time_ms": 5.819774094,
      "filtered": 100,
      "r_filtered": 0.002553496,
      "index_condition": "OT.o_orderDATE between '1998-06-01' and '1998-12-06'",
      "attached_condition": "OT.o_totalprice > 0.9 * (subquery#2)"
    },
    "subqueries": [
      {
        "expression_cache": {
          "state": "disabled",
          "r_loops": 200,
          "r_hit_ratio": 0,
          "query_block": {
            "select_id": 2,
            "r_loops": 39162,
            "r_total_time_ms": 1032.541544,
            "outer_ref_condition": "OT.o_custkey is not null",
            "table": {
              "table_name": "orders",
              "access_type": "ref",
              "possible_keys": ["i_o_custkey"],
              "key": "i_o_custkey",
              "key_length": "5",
              "used_key_parts": ["o_custkey"],
              "ref": ["dbt3sf1.OT.o_custkey"],
              "r_loops": 39162,
              "rows": 7,
              "r_rows": 17.73660691,
              "r_table_time_ms": 985.8381759,
              "r_other_time_ms": 32.85006493,
              "filtered": 100,
              "r_filtered": 100
            }
          }
}]}}
```

Sorting

```
analyze format=json
select * from customer
order by c_acctbal desc limit 50
```

```
"query_block": {
  "select_id": 1,
  "r_loops": 1,
  "r_total_time_ms": 52.68082832,
  "read_sorted_file": {
    "r_rows": 50,
    "filesort": {
      "sort_key": "customer.c_acctbal desc",
      "r_loops": 1,
      "r_total_time_ms": 52.65527147,
      "r_limit": 50,
      "r_used_priority_queue": true,
      "r_output_rows": 51,
      "r_sort_mode": "sort_key,addon_fields",
      "table": {
        "table_name": "customer",
        "access_type": "ALL",
        "r_loops": 1,
        "rows": 148473,
        "r_rows": 150000,
        "r_table_time_ms": 35.72704671,
        "r_other_time_ms": 16.90903666,
        "filtered": 100,
        "r_filtered": 100
      }
    }
  }
}
```

- EXPLAIN shows “Using filesort”
  - Will it really read/write files?
  - Is my @@sort_buffer_size large enough?
  - Is priority queue optimization used?

Sorting (2)

```
analyze format=json
select * from customer
order by c_acctbal desc limit 50
```

```
"query_block": {
  "select_id": 1,
  "r_loops": 1,
  "r_total_time_ms": 52.68082832,
  "read_sorted_file": {
    "r_rows": 50,
    "filesort": {
      "sort_key": "customer.c_acctbal desc",
      "r_loops": 1,
      "r_total_time_ms": 52.65527147,
      "r_limit": 50,
      "r_used_priority_queue": true,
      "r_output_rows": 51,
      "r_sort_mode": "sort_key,addon_fields",
      "table": {
        "table_name": "customer",
        "access_type": "ALL",
        "r_loops": 1,
        "rows": 148473,
        "r_rows": 150000,
        "r_table_time_ms": 35.72704671,
        "r_other_time_ms": 16.90903666,
        "filtered": 100,
        "r_filtered": 100
      }
    }
  }
}
```

```
analyze format=json
select * from customer
order by c_acctbal desc limit 50000
```

```
"query_block": {
  "select_id": 1,
  "r_loops": 1,
  "r_total_time_ms": 85.70263992,
  "read_sorted_file": {
    "r_rows": 50000,
    "filesort": {
      "sort_key": "customer.c_acctbal desc",
      "r_loops": 1,
      "r_total_time_ms": 79.33072276,
      "r_limit": 50000,
      "r_used_priority_queue": false,
      "r_output_rows": 150000,
      "r_sort_passes": 1,
      "r_buffer_size": "2047Kb",
      "r_sort_mode": "sort_key,packed_addon_fields",
      "table": {
        "table_name": "customer",
        "access_type": "ALL",
        "r_loops": 1,
        "rows": 148473,
        "r_rows": 150000,
        "r_table_time_ms": 37.21097011,
        "r_other_time_ms": 42.09169676,
        "filtered": 100,
        "r_filtered": 100
      }
    }
  }
}
```

ANALYZE and UPDATEs

- ANALYZE works for any statement that supports EXPLAIN
  - UPDATE, DELETE
- DML statements will do the modifications!
  - ANALYZE UPDATE ... will update
  - ANALYZE DELETE ... will delete
  - (PostgreSQL’s EXPLAIN ANALYZE also works like this)
- Don’t want modifications to be made?

```
begin;
analyze update ...;
rollback;
```

Overhead of ANALYZE

- ANALYZE data is:
  - Counters (cheap, always counted)
  - Time measurements (expensive. counted only if running ANALYZE)
- Max. overhead I observed: 10%
  - Artificial example constructed to maximize overhead
  - Typically less

ANALYZE and slow query log

- my.cnf

```
slow_query_log=ON
log_slow_verbosity=explain
...
```

- hostname-slow.log

```
# Time: 200817 12:12:47
# User@Host: root[root] @ localhost []
# Thread_id: 18  Schema: dbt3sf1  QC_hit: No
# Query_time: 3.948642  Lock_time: 0.000062  Rows_sent: 0  Rows_examined: 7501215
# Rows_affected: 0  Bytes_sent: 1756
#
# explain: id   select_type     table   type    possible_keys   key     key_len ref     rows    r_rows  filtered
r_filtered      Extra
# explain: 1    SIMPLE  orders  ALL     PRIMARY,i_o_orderdate   NULL    NULL    NULL    1492405 1500000.00      50.00   100.00
Using where
# explain: 1    SIMPLE  lineitem        ref     PRIMARY,i_l_orderkey,i_l_orderkey_quantity      PRIMARY 4
dbt3sf1.orders.o_orderkey       24.00     100.00  0.00    Using where
#
SET timestamp=1597655567;
select *
...
```

- **log_slow_verbosity=explain** shows ANALYZE output
- Obstacles to printing JSON:
  - No timing data
  - Keeping format compatible with pt-query-digest?

ANALYZE FORMAT=JSON Summary

- It is EXPLAIN FORMAT=JSON with execution data
- Use **r_rows** and **r_filtered** to check estimates vs reality
- Use **r_total_time_ms** and **r_loops** to narrow down problems
- Sorting, buffering, caches report **r_** values about their execution.
- Please do attach ANALYZE FORMAT=JSON output when asking for optimizer help or reporting an issue.

MySQL 8:

EXPLAIN ANALYZE

EXPLAIN ANALYZE in MySQL 8

- Introduced in MySQL 8.0.18 (Oct, 2019)
- Produces PostgreSQL-like output
  - (In next slides, I will edit it for readability)

```
| -> Nested loop inner join  (cost=642231.45 rows=1007917) (actual time=65.871..4580.448 rows=24 loops=1)
    -> Filter: (orders.o_orderDATE between '1990-01-01' and '1998-12-06')  (cost=151912.95 rows=743908) (actual
time=0.154..536.302 rows=1500000 loops=1)
        -> Table scan on orders  (cost=151912.95 rows=1487817) (actual time=0.146..407.074 rows=1500000 loops=1)
    -> Filter: (lineitem.l_extendedprice > 104500)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=0
loops=1500000)
        -> Index lookup on lineitem using PRIMARY (l_orderkey=orders.o_orderkey)  (cost=0.25 rows=4) (actual
time=0.002..0.002 rows=4 loops=1500000)
```

- Reason for this: the runtime is switched to “Iterator Interface”.

Basics

```
explain analyze
select * from orders
where
  o_orderdate between '1998-06-01' and '1998-12-06' and
  o_totalprice>50000
```

```
| -> Filter: (orders.o_totalprice > 50000)  (cost=35418.86 rows=26233)
        (actual time=0.607..51.881 rows=33705 loops=1)
    -> Index range scan on orders using i_o_orderdate, with index
       condition: (orders.o_orderDATE between '1998-06-01' and '1998-12-06')
       (cost=35418.86 rows=78708)
       (actual time=0.603..50.277 rows=39162 loops=1)
```

- **loops** is like MariaDB's **r_loops**
- **rows** is number of rows in the output
- **rows** after Filter / **rows** before it = 33705/39162=86% - this is **r_filtered**.

- **actual time** shows **X..Y**
  - X - time to get the first tuple
  - Y- time to get all output
  - Both are averages across all loops.

A complex query: join

```
analyze format=json
select *
from
  lineitem, orders
where
  o_orderkey=l_orderkey and
  o_orderdate between '1990-01-01'
                  and '1998-12-06' and
  l_extendedprice > 1000000
```

```
-> Nested loop inner join  (cost=642231.45 rows=1007917)
                      (actual time=4496.131..4496.131 rows=0 loops=1)
  -> Filter: (orders.o_orderDATE between '1990-01-01' and '1998-12-06')
             (cost=151912.95 rows=743908)
             (actual time=0.162..534.395 rows=1500000 loops=1)
     -> Table scan on orders  (cost=151912.95 rows=1487817)
                (actual time=0.154..402.573 rows=1500000 loops=1)
  -> Filter: (lineitem.l_extendedprice > 1000000)  (cost=0.25 rows=1)
             (actual time=0.003..0.003 rows=0 loops=1500000)
     -> Index lookup on lineitem using PRIMARY
              (l_orderkey=orders.o_orderkey)
              (cost=0.25 rows=4)
              (actual time=0.002..0.002 rows=4 loops=1500000)
```


```
"query_block": {
  "select_id": 1,
  "r_loops": 1,
  "r_total_time_ms": 4056.659499,
  "table": {
    "table_name": "orders",
    "access_type": "ALL",
    "possible_keys": ["PRIMARY", "i_o_orderdate"],
    "r_loops": 1,
    "rows": 1492405,
    "r_rows": 1500000,
    "r_table_time_ms": 323.3849353,
    "r_other_time_ms": 118.576661,
    "filtered": 49.99996567,
    "r_filtered": 100,
    "attached_condition": "orders.o_orderDATE between '1990-
  },
  "table": {
    "table_name": "lineitem",
    "access_type": "ref",
    "possible_keys": ["i_l_orderkey", "i_l_orderkey_quantity"
    "key": "PRIMARY",
    "key_length": "4",
    "used_key_parts": ["l_orderkey"],
    "ref": ["dbt3sf1.orders.o_orderkey"],
    "r_loops": 1500000,
    "rows": 2,
    "r_rows": 4.00081,
    "r_table_time_ms": 3454.758327,
    "r_other_time_ms": 159.9274175,
    "filtered": 100,
    "r_filtered": 0,
    "attached_condition": "lineitem.l_extendedprice > 1000000
  }
}
```


- **r_total_time_ms** = actual_time.second_value * loops. Practice your arithmetics skills :-)

Sorting

```
explain analyze select * from customer order by c_acctbal desc limit 50
```

```
| -> Limit: 50 row(s)  (actual time=55.830..55.837 rows=50 loops=1)
    -> Sort: customer.c_acctbal DESC, limit input to 50 row(s) per chunk  (cost=15301.60 rows=148446)
             (actual time=55.830..55.835 rows=50 loops=1)
        -> Table scan on customer  (actual time=0.137..38.462 rows=150000 loops=1)
```

```
explain analyze select * from customer order by c_acctbal desc limit 50000
```

```
| -> Limit: 50000 row(s)  (actual time=101.536..106.927 rows=50000 loops=1)
    -> Sort: customer.c_acctbal DESC, limit input to 50000 row(s) per chunk  (cost=15301.60 rows=148446)
             (actual time=101.535..105.539 rows=50000 loops=1)
        -> Table scan on customer  (actual time=0.147..37.550 rows=150000 loops=1)
```

- No difference
- But look at Optimizer Trace: it does show the algorithm used:

```
"filesort_priority_queue_optimization": {
  "limit": 50,
  "chosen": true
},
```

```
"filesort_priority_queue_optimization": {
  "limit": 50000
},
"filesort_execution": [
],
"filesort_summary": {
  "max_rows_per_buffer": 303,
  "num_rows_estimate": 1400441,
  "num_rows_found": 150000,
  "num_initial_chunks_spilled_to_disk": 108,
  "peak_memory_used": 270336,
  "sort_algorithm": "std::stable_sort",
}
```

Handling for UPDATE/DELETE

- Single-table UPDATE/DELETE is not supported

```
mysql> explain analyze delete from orders where o_orderkey=1234;
+------------------------------------------+
| EXPLAIN                                  |
+------------------------------------------+
| <not executable by iterator executor>
 |
+------------------------------------------+
1 row in set (0.00 sec)
```

  - The same used to be true for outer joins but was fixed

- Multi-table UPDATE/DELETE is supported
  - Unlike in MariaDB/PostgreSQL, won’t make any changes.

MySQL's EXPLAIN ANALYZE summary

- The syntax is EXPLAIN ANALYZE
- Postgres-like output
- Not all plan details are shown
  - Some can be found in the Optimizer Trace
- Not all statements are supported
- Can have higher overhead
  - Artificial "bad" example: 50% overhead (vs 10% in MariaDB)
- Shows extra info:
  - Has nodes for grouping operations (worth adding?)
  - #rows output for each node (worth adding, too?)
  - "Time to get the first row" (not sure if useful?)

Final conclusions

- MariaDB has
  - ANALYZE <statement>
  - ANALYZE FORMAT=JSON <statement>
- Stable feature
- Very useful in diagnosing optimizer issues
- MySQL has got EXPLAIN ANALYZE last year
  - A similar feature
  - Different interface, printed data, etc
  - No obvious advantages over MariaDB’s?

Thanks!