New query engine features in MariaDB

Sergey Petrunya, Monty Program Ab
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Predicate caching
  - FROM subquery optimizations
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Predicate caching
  - FROM subquery optimizations
Table elimination

- Applicable for queries over normalized data
- Present in “big” databases like Oracle, SQL Server
  - And will be in PostgreSQL 9.0 (they call it “join removal”)
- Basic idea
  *Detect outerjoins that have “unused” inner sides and delete those inner sides*

```sql
SELECT tbl1.*
FROM
  tbl1 LEFT JOIN tbl2 ON tbl2.primary_key=tbl1.id
WHERE
  condition(tbl1.*)
```
- It is guaranteed that for each record of tbl1
  - tbl2 will have not more than one match (tbl2.primary_key=..)
  - If tbl2 has no match, LEFT JOIN will generate a NULL-record
The case for table elimination

- Highly-normalized data:
  actor(name, date_of_birth, rating)
  is stored as:

```sql
create table ac_anchor(AC_ID int primary key);
create table ac_name(AC_ID int, ACNAM_name char(N), primary key(AC_ID));
create table ac_dob(AC_ID int, ACDOB_birthdate date, primary key(AC_ID));
create table ac_rating(AC_ID int, ACRAT_rating int, ACRAT_fromdate date, primary key(AC_ID, ACRAT_fromdate));
```
The case for table elimination (2)

Then select back:

```sql
create view actors as select * from
select
    ac_anchor.AC_ID, ACNAM_Name, ACDOB_birthdate, ACRAT_rating
from
    ac_anchor
left join ac_name on ac_anchor.AC_ID=ac_name.AC_ID
left join ac_dob on ac_anchor.AC_ID=ac_dob.AC_ID
left join ac_rating on (ac_anchor.AC_ID=ac_rating.AC_ID and
ac_rating.ACRAT_fromdate =
    (select max(sub.ACRAT_fromdate) from ac_rating sub
     where sub.AC_I=ac_rating.AC_ID))

select ACRAT_rating from actors where ACNAM_name='Gary Oldman';
```
### Table elimination – examples

**Explain select** `ACRAT_rating, ACDOB_birthdate` **from actors where** `ACNAM_name='Gary Oldman';` **;**

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_anchor</td>
<td>index</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_name</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_rating</td>
<td>ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
<tr>
<td>3</td>
<td>DEPENDENT SUBQUERY</td>
<td>sub</td>
<td>ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_rating.AC_ID</td>
</tr>
</tbody>
</table>

**Explain select** `ACRAT_rating` **from actors where** `ACNAM_name='Gary Oldman';` **;**

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_anchor</td>
<td>index</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_name</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_rating</td>
<td>ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
<tr>
<td>3</td>
<td>DEPENDENT SUBQUERY</td>
<td>sub</td>
<td>ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_rating.AC_ID</td>
</tr>
</tbody>
</table>

**Explain select** `ACDOB_birthdate` **from actors where** `ACNAM_name='Gary Oldman';` **;**

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_anchor</td>
<td>index</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_name</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY</td>
<td>ac_dob</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>ac_anchor.AC_ID</td>
</tr>
</tbody>
</table>
Table elimination benchmark

- Not a lot of gains with lookups on primary key
- For DBT-3 data (no blobs, no historized data):

- With historized data: 2-3-4x benefit (depending on how much history)
Table elimination – conclusion

- With table elimination, you can:
  - Do normalization on optional/historic data
  - Create a view with LEFT JOINs that presents denormalized view of the data
  - Use this view and get something like “index only” scans when you're accessing only some attributes
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
  - Join buffering now works with outer joins
- Index Condition Pushdown (MariaDB 5.3)
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Predicate caching
  - FROM subquery optimizations
Batched Key Access – idea

- Background: nested loops join execution

```sql
mysql> select * from Country, City
```

<table>
<thead>
<tr>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>ref</td>
<td>PRIMARY,Continent</td>
<td>Continent</td>
<td>1</td>
<td>const</td>
<td>37</td>
<td>Using index condition</td>
</tr>
<tr>
<td>City</td>
<td>ref</td>
<td>CountryCode</td>
<td>CountryCode</td>
<td>3</td>
<td>Country.Code</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>
Batched Key Access – idea

- Batched Key access execution

```sql
mysql> select * from City, Country
```

<table>
<thead>
<tr>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>ref</td>
<td>PRIMARY,Continent</td>
<td>Continent</td>
<td>1</td>
<td>const</td>
<td>37</td>
<td>Using index condition</td>
</tr>
</tbody>
</table>

![Diagram showing batched key access execution](attachment:diagram.png)
Why use Batched Key Access

- 1. Storage engine can optimize record reads when scanning many ranges at once
- 2. Buffering allows not to make the same lookup multiple times
BKA advantage #1

- InnoDB/MyISAM/Maria read table rows in disk order
- NDB Cluster reduces #roundtrips
BKA Advantage #2

- Buffering allows not to make the same lookup multiple times

```sql
select * from supplier, customer
where supplier.city=customer.city and ...
```
Batched Key Access properties

- Initial implementation done at Sun/MySQL for MySQL 6.0
- MariaDB:
  - Backport from MySQL 6.0 to MariaDB 5.3
  - Infamous InnoDB's + ICP/BKA bugs fixed
- Benchmark
  - DBT-3 scale=10, Query #3.1 (enumerates about 6M rows), IO-bound

```sql
select l_orderkey, sum(l_extendedprice * (1-l_discount)) as revenue, o_orderdate, o_shippriority
from customer, orders, lineitem
where c_mktsegment = 'AUTOMOBILE' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= '1994-07-01' and o_orderdate < '1995-01-01' and l_shipdate > '1995-01-14'
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 5;
```

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>No BKA</td>
<td>1 hour 4 min 37 sec</td>
<td>1 x</td>
</tr>
<tr>
<td>BKA, 1M buffer</td>
<td>10 min 25 sec</td>
<td>6.2 x</td>
</tr>
<tr>
<td>BKA, 10M buffer</td>
<td>6 min 16 sec</td>
<td>10.31 x</td>
</tr>
</tbody>
</table>
DBT-3, scale=10, Query#3.2, IO-bound load

```
explain
select l_orderkey, sum(l_extendedprice * (1-l_discount)) as revenue,
       o_orderdate, o_shippriority
from customer, orders, lineitem
where c_nationkey = 12 and c_custkey = o_custkey and
       l_orderkey = o_orderkey and l_shipdate > '1995-01-14'
group by l_orderkey, o_orderdate, o_shippriority
order by revenue desc, o_orderdate
limit 5;
```

<table>
<thead>
<tr>
<th></th>
<th>No BKA</th>
<th>2 hours 16 min 4 sec</th>
<th>1 x</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKA, 1M buffer</td>
<td>2 min 41 sec</td>
<td></td>
<td>50.71 x</td>
</tr>
<tr>
<td>BKA, 10M buffer</td>
<td>2 min 36 sec</td>
<td></td>
<td>52.33 x</td>
</tr>
</tbody>
</table>

General observation from CPU-bound runs

- 2x-10x improvement
Join buffering and outer joins

- It is not possible to use join buffering for outer joins in MySQL 5.x
- It was a low-hanging fruit when you have BKA code
- So it's there in MariaDB 5.3
  - Outer joins will use BKA when using key access
  - And will use join buffering with when not using key access
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
  - Join buffering now works with outer joins
- Index Condition Pushdown (MariaDB 5.3)
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
- **The idea:**
  
  *When using index-based access method, check condition on index columns before fetching the table record*
The idea:
When using index-based access method, check condition on index columns before fetching the table record
Index Condition Pushdown

- The idea:
  *When using index-based access method, check condition on index columns before fetching the table record*
Index Condition Pushdown

- Works with any index-based access method: range, ref, eq_ref, BKA
- Original implementation: MySQL 6.0
  - Infamous for bugs in Index Condition Pushdown + InnoDB
- MariaDB
  - 6.0 code backported to MariaDB 5.3
  - Problems with innodb are believed to be fixed
    - If you're a long timer and willing to check:
      ```
      SET engine_condition_pushdown=on|off;
      no longer controls Index Condition Pushdown, use
      SET @@optimizer_switch= 'index_condition_pushdown=on|off'
      instead
      ```
DBT-3 data, scale=1

```sql
alter table lineitem
    add index s_r (l_shipdate, l_receiptdate);

select count(*) from lineitem
where
    l_shipdate between '1993-01-01' and '1993-01-01' + interval 30 day and
    datediff(l_receiptdate,l_shipdate) > 25 and
    l_quantity > 40;
```

Results

- Cold caches: from 5 min down to 1 min
- Hot caches: from 0.19 sec down to 0.07 sec
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
Semi-join subqueries

- A class of subqueries of greatest practical importance

```
SELECT ...
FROM outer_tables
WHERE
    something IN (SELECT something else FROM inner_tables WHERE ... ) AND ...
```

- MySQL 4.1/5.0: outer-to-inner execution only.

```
SELECT * FROM people WHERE name IN (SELECT owner FROM big_yachts WHERE cost > 50M)
```
Semi-join subqueries: MySQL 6.0/MariaDB

- MySQL 6.0/MariaDB: inside-to-outside execution is supported

```sql
SELECT * FROM people WHERE name IN (SELECT owner FROM big_yachts WHERE cost > 50M)
```
Insideout execution modes

```sql
SELECT * FROM people WHERE name IN (SELECT owner FROM big_yachts WHERE cost > 50M)
```

1. Remove duplicates before joining:
   - Materialization
   - Loose scan

2. Remove duplicates after joining:
   - Duplicate Weedout
Subquery optimizations summary

- WHERE ... IN (SELECT w/o grouping) subqueries are handled with a comprehensive set of optimizations

<table>
<thead>
<tr>
<th></th>
<th>Outer-to-inner</th>
<th>Inner-to-outer</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Match</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Loose Scan</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Materialization</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Duplicate Weedout</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

- Outer-to-inner:
  - First Match
  - Loose Scan
  - Materialization
  - Duplicate Weedout

- Inner-to-outer:
  - First Match
  - Loose Scan
  - Materialization
  - Duplicate Weedout
Quoting previous results with MySQL 6.0:

- MySQL bugs/customer issues that are easily repeatable
  - Found 10 subquery cases
  - Taking PostgreSQL's speed as 1.0:

<table>
<thead>
<tr>
<th>No 6.0 optimizations</th>
<th>Materialization</th>
<th>Semi-join</th>
</tr>
</thead>
<tbody>
<tr>
<td>67285.714</td>
<td>34.286</td>
<td>1.429</td>
</tr>
<tr>
<td>59490.000</td>
<td>780.000</td>
<td>n/a</td>
</tr>
<tr>
<td>9.477</td>
<td>2.109</td>
<td>0.004</td>
</tr>
<tr>
<td>151.429</td>
<td>206.667</td>
<td>0.476</td>
</tr>
<tr>
<td>1360.000</td>
<td>490.000</td>
<td>10.000</td>
</tr>
<tr>
<td>670.453</td>
<td>0.264</td>
<td>1.052</td>
</tr>
<tr>
<td>16.364</td>
<td>0.455</td>
<td>0.182</td>
</tr>
<tr>
<td>10.000</td>
<td>0.625</td>
<td>n/a</td>
</tr>
<tr>
<td>5648.649</td>
<td>3.243</td>
<td>0.270</td>
</tr>
<tr>
<td>962.500</td>
<td>1.500</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>816.48</strong></td>
<td><strong>2.68</strong></td>
<td><strong>0.48</strong></td>
</tr>
</tbody>
</table>

Medians:
- MariaDB's time should be MIN(column2, column3), or MAX(column2, column3)
  - Can't re-run because data is Sun-private

Run parameters
- MySQL 6.0.3
- PostreSQL 8.3.0
- No tuning, all default settings
- Small query population
- => numbers only show order of magnitude
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
NULL-aware materialization

- With subqueries, SQL's semantics for NULL value is “unknown”.

- This gives us:

  NULL IN (SELECT ... FROM something) = NULL
  NULL IN (SELECT ... FROM no_data) = FALSE

  'foo' IN (SELECT ... FROM
             'bar'     ) = TRUE
  NULL

  'foo' IN (SELECT ... FROM
             'baz'     ) = NULL
  NULL

- An important special case where we don't care whether result is NULL or FALSE:

  ... WHERE smth IN (SELECT ...) AND|OR ...

- In all other cases, including WHERE ... NOT IN(SELECT...) we do care and need correct evaluation
NULL-aware materialization (1)

- Materialization strategy without NULLs:

SELECT ... FROM requests
WHERE (car_model, state) NOT IN (SELECT car_model, state FROM offers
WHERE offer_date=NOW())
- NULLs outside

SELECT ... FROM requests
WHERE (car_model, state) NOT IN (SELECT car_model, state FROM offers
   WHERE offer_date=NOW())

- If we get NULL on the left of IN, we must search for partial match
  - Generally, hash indexes used for materialization table do not support partial matching, so one has to do full scan on the temptable
Now, NULLs inside the subquery

SELECT ... FROM requests
WHERE (car_model, state) NOT IN (SELECT car_model, state FROM offers
WHERE offer_date=NOW())

- WV Bug in CO? No
- WV Bug at unspecified location? No
- Unknown model of car in CO? Yes

Subquery evaluates to NULL

- NULLs on the inside of subquery also require partial match searches
Good news

- MariaDB 5.3 will include advanced materialization/lookup strategy that will
  - Support handling of NULLs inside/outside of the subqueries
  - Has no overhead over regular materialization if columns are NULLable but no NULLs were encountered
  - No significant degradation for big fractions of NULLs on either side

```sql
select count(*) from customer
where (c_custkey, c_pref_nationkey_05, c_pref_brand_05) NOT IN
  (select o_custkey, s_nationkey, p_brand
   from orders, supplier, part, lineitem
   where l_orderkey = o_orderkey and
     l_suppkey = s_suppkey and
     l_partkey = p_partkey and
     p_retailprice < 1200 and
     l_shipdate >= '1996-04-01' and l_shipdate < '1996-04-05' and
     o_orderdate >= '1996-04-01' and o_orderdate < '1996-04-05');
```

<table>
<thead>
<tr>
<th></th>
<th>IN-&gt;EXISTS</th>
<th>Materialization</th>
<th>Times faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBT3 scale=1, cold run</td>
<td>34</td>
<td>30</td>
<td>1.13</td>
</tr>
<tr>
<td>DBT3 scale=1, hot run</td>
<td>3.55</td>
<td>0.27</td>
<td>13.15</td>
</tr>
<tr>
<td>DBT3 scale=10, cold run</td>
<td>532</td>
<td>712</td>
<td>0.75</td>
</tr>
<tr>
<td>DBT3 scale=10, hot run</td>
<td>40</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>
Not so good news

- Experiments show that
  - There are cases when Materialization is much better
  - There are cases where subquery re-execution (old approach) is much better
    - Roughly speaking: when one of the tables already has an index that materialization would create
  - Will need to work on a cost-based decision between materialization and subquery re-execution
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
Materialization-scan for grouping queries

- Detected while analyzing user cases

```sql
select * from customer
where customer_id in (select customer_id
from orders
group by customer_id
having sum(cost) > 1M)
```

- MySQL 5.x/6.0:

  - outer_query
  - subquery

- MariaDB 5.3

  - subquery
  - outer_query

Big table

Table with a few groups
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
Subquery value caching

- The idea: take any correlated subquery

```sql
select ...
from outer_table_1, outer_table_2
where/having
  ... (select from inner_table
        where outer_table_1.column1=... and
        ...
        inner_column < outer_table_2.column2)
```

- Subquery value is a function of \((\text{outer_table}_1.\text{column1}, \text{outer_table}_2.\text{column2})\).

- Generally, the subquery is evaluated many times for different values of \(\text{outer_table}_1.\text{column1}\) and \(\text{outer_table}_2.\text{column2}\)
  - But values do repeat

- Create/use a cache

\((\text{outer_table}_1.\text{column1}, \text{outer_table}_2.\text{column2}) \rightarrow \text{subquery_result}\)
What exactly is being cached

- The cache will hold scalars for all kinds of subqueries:

  EXISTS (SELECT * FROM promotions WHERE
  state= outer_table.state OR
  industry= outer_table.industry)

  (outer_table.state, outer_table.industry) -> TRUE/FALSE

  SELECT * FROM brands as outer_tbl
  WHERE (SELECT avg(discount) FROM sales WHERE brand=
  outer_tbl.brandname) > 0.25

  outer_tbl.brandname -> avg(discount)

  outer_table.brand_name IN (SELECT brand_name FROM promotions WHERE
  state=outer_table.state)

  (outer_table.brand_name, outer_table.state) -> TRUE/FALSE/NULL
Subquery value cache implementation

- HEAP temporary table is used as hashtable
- The cache is of infinite size
- Convert to MyISAM/Maria if HEAP table overflows
  - MyISAM lookups are slow but still faster than subquery re-executions
- Use of cache can be controlled from SQL with
  `SET optimizer_switch='subquery_cache=on|off'`
Subquery cache benchmark

- DBT-3 data, scale 1.

- Query #17, total revenue on low-quantity orders

```sql
select sum(l_extendedprice) / 7.0 as avg_yearly
from lineitem, part
where
    p_partkey = l_partkey and
    p_brand = 'Brand#42' and p_container = 'JUMBO BAG' and
    l_quantity < (select 0.2 * avg(l_quantity) from lineitem
                  where l_partkey = p_partkey) ;
```

2.3x speedup, from 3.28 to 1.34 sec, cache hit ratio: 96%

- Own query: customers with balance near top in their nation

```sql
select count(*) from customer
where c_acctbal > 0.8 * (select max(c_acctbal) 
                          from customer C 
                          where C.c_nationkey=customer.c_nationkey 
                          group by c_nationkey);
```

2640 times speedup, from 43 min to 0.69 sec.
New query engine features

- Table elimination (MariaDB 5.1)
- Batched Key Access (MariaDB 5.3)
- Join buffering now works with outer joins
- Index Condition Pushdown
- Subquery optimizations (MariaDB 5.3)
  - Backport of 6.0 features
  - NULL-aware materialization
  - Materialization-scan for grouping queries
  - Subquery value caching
  - FROM subquery optimizations
Long-time MySQL problem:

```sql
select * from
(select * from flights where flight_type<>'Military') as civil_flight,
train_departure
where
train_departure.hour = flight.arrival_hour;
```

Worse: EXPLAIN does materialization too, so will be very slow, too.

Can work around with `CREATE VIEW civil_flight`
FROM subquery optimizations(2)

- Initial work started at Sun as WL#3485, never finished
- Backported and feature-finished in MariaDB 5.3

```sql
select *
from
(select * from flights where flight_type<>'Military') as civil_flight,
train_departure
where
train_departure.hour = flight.arrival_hour;
```

**Diagram:**
- `flights`
- `train_departures`
- `Civil flight?`

*We don't have to materialize, so can access to original flights table, including its indexes*
Benchmark for FROM subquery optimizations

- DBT-3, scale=1, “hot”, query#8

```sql
select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume) as mkt_share
from
(select substr(o_orderdate, 1, 4) as o_year,
    l_extendedprice * (1 - l_discount) as volume,
    n2.n_name as nation
from part, supplier, lineitem, orders, customer,
    nation n1, nation n2, region
where p_partkey = l_partkey and s_suppkey = l_suppkey and
    l_orderkey = o_orderkey and o_custkey = c_custkey and
    c_nationkey = n1.n_nationkey and n1.n_regionkey = r_regionkey and
    r_name = 'AMERICA' and s_nationkey = n2.n_nationkey and
    o_orderdate between '1995-01-01' and '1996-12-31' and
    p_type = 'ECONOMY ANODIZED STEEL'
) as all_nations
group by o_year
order by o_year;
```

**EXPLAIN execution time**

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>46.91 sec</td>
<td>0.01 sec</td>
</tr>
</tbody>
</table>

**Query execution time**

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>47.06 sec</td>
<td>10.46 sec</td>
</tr>
</tbody>
</table>

4.5x speedup
**DBT-3, scale=1, “hot”, query#9**

```sql
select nation, o_year, sum(amount) as sum_profit
from
(
    select n_name as nation, substr(o_orderdate, 1, 4) as o_year,
         l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity
         as amount
    from part, supplier, lineitem, partsupp, orders, nation
    where s_suppkey = l_suppkey and ps_suppkey = l_suppkey and
    ps_partkey = l_partkey and p_partkey = l_partkey and
    o_orderkey = l_orderkey and s_nationkey = n_nationkey and
    p_name like '%green%'
) as profit
group by nation, o_year
order by nation, o_year desc
limit 5;
```

**EXPLAIN execution time**

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>2 min 26 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>After</td>
<td>0.02 sec</td>
<td></td>
</tr>
</tbody>
</table>

**Query execution time**

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>2 min 36 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>After</td>
<td>53 sec</td>
<td></td>
</tr>
</tbody>
</table>

2.9x speedup
Summary

- MariaDB 5.1
  - Table elimination

- MariaDB 5.2
  - 

- MariaDB 5.3
  - Batched Key Access
  - Join buffering now works with outer joins
  - Index Condition Pushdown
  - Subquery optimizations (MariaDB 5.3)
    - Backport of 6.0 optimizations
    - NULL-aware materialization
    - Materialization-scan for grouping queries
    - Subquery value caching
    - FROM subquery optimizations

Highly normalized data, outer joins

* Big joins,
* Big range scans on composite indexes
* Subqueries
The End

Q & A