Why Are The New Optimizer Features Important and How Can I Benefit From Them?

Igor Babaev  igor@askmonty.org
Sergey Petrunya psergey@askmonty.org
New query optimizer features

- We'll talk about
  - Fresh finished query optimizer features in MariaDB

- General development theme
  - Big data (IO-bound loads)
  - Big queries
    - Joins
    - SELECTs that retrieve lots of rows
New query optimizer features

- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- Batched Key Access [improvements]
- Hash join
Benchmarking setup

- DBT-3 scale=10
  - XtraDB storage engine
  - 28 GB total dataset
  - Biggest table: 'lineitem' 60M rows, 22G on disk

- 'ontime': table form transtats.bts.gov dataset
  - XtraDB storage engine
  - Subset: flights in January-April 2009
  - 1.5 M rows, 832MB.

- innodb_buffer_pool=256M unless said otherwise
New query optimizer features

- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- Batched Key Access [improvements]
- Hash join
index_merge vs range optimization

- A long-known, much-complained problem in MySQL:

```sql
MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA');
+----+----------+-----------------+-------------------+-------------+-----------+-------+----+-----+-------------------------------------+
<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>index_merge</td>
<td>Origin,Dest</td>
<td>Origin,Dest</td>
<td>6,6</td>
<td>NULL</td>
<td>92850</td>
<td>Using union(Origin,Dest); Using where</td>
</tr>
</tbody>
</table>

MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and securitydelay=0;
+----+----------+-----------------+-------------------+-------------+-----------+-------+----+-----+-------------------------------------+
<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>ref</td>
<td>Origin,Dest,SecurityDelay</td>
<td>SecurityDelay</td>
<td>5</td>
<td>const</td>
<td>791546</td>
<td>Using where</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MySQL [ontime]> explain select * from ontime where (Origin='SEA' or Dest='SEA') and depdelay < 12*60;
+----+----------+-----------------+-------------------+-------------+-----------+-------+----+-----+-------------------------------------+
<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>ALL</td>
<td>Origin,DepDelay,Dest</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1583093</td>
<td>Using where</td>
</tr>
</tbody>
</table>
```

- Additional AND-ed predicates cause index_merge plan to be removed from consideration, and a worse plan to be chosen
  - Slowdown can be 10x, 100x, ...

10x more. Ouch!

15x more. Ouch!
Now, let's try the same in MariaDB 5.3:

```
MariaDB [ontime]>
explain select * from ontime where (Origin='SEA' or Dest='SEA');
```

```
+----+-----------+------+-----------+-------------+-----------+-------+----+-----+-------------------------------------+
| id | select_type| table | type      | possible_keys| key        | key_len| ref | rows |Extra                                |
+----+-----------+------+-----------+-------------+-----------+-------+----+-----+-------------------------------------+
|  1 | SIMPLE    | ontime| index_merge| Origin,Dest  | Origin,Dest| 6,6    | NULL| 92800|Using union(Origin,Dest); Using where|
+----+-----------+------+-----------+-------------+-----------+-------+----+-----+-------------------------------------+

MariaDB [ontime]>
explain select * from ontime where (Origin='SEA' or Dest='SEA') and depdelay < 12*60;
```

```
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+
| id | select_type| table | type      | possible_keys       | key        | key_len| ref | rows |Extra                                |
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+
|  1 | SIMPLE    | ontime| index_merge| Origin,DepDelay,Dest| Origin,Dest| 6,6    | NULL| 92800|Using union(Origin,Dest); Using where|
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+

MariaDB [ontime]>
explain select * from ontime where (Origin='SEA' or Dest='SEA') and securitydelay=0;
```

```
+----+-----------+------+-----------+-------------------------------+-----------+-------+----+-----+-------------------------------------+
| id | select_type| table | type      | possible_keys            | key        | key_len| ref | rows |Extra                                |
+----+-----------+------+-----------+-------------------------------+-----------+-------+----+-----+-------------------------------------+
|  1 | SIMPLE    | ontime| index_merge| Origin,Dest,SecurityDelay    | Origin,Dest| 6,6    | NULL| 92800|Using union(Origin,Dest); Using where|
+----+-----------+------+-----------+-------------------------------+-----------+-------+----+-----+-------------------------------------+

MariaDB [ontime]>
explain select * from ontime where (Origin='SEA' or Dest='SEA') and depdelay < 12*60;
```

```
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+
| id | select_type| table | type      | possible_keys       | key        | key_len| ref | rows |Extra                                |
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+
|  1 | SIMPLE    | ontime| index_merge| Origin,DepDelay,Dest | Origin,Dest| 6,6    | NULL| 92800|Using union(Origin,Dest); Using where|
+----+-----------+------+-----------+--------------------+-----------+-------+----+-----+-------------------------------------+
```

Will still use index_merge

Same here
We call it

"Fair choice between range and index_merge optimizations"

- It is in MariaDB 5.3
- New functionality is always on (no way to switch to old behavior)
- No known problems.
New query optimizer features

- Correct optimization of index_merge vs range
- `index_merge/sort-intersect`
- Batched Key Access
- Hash join
MySQL and MariaDB 5.{1,2} support index_merge/intersection:

MySQL [ontime] > explain select avg(arrdelay) from ontime where depdel15=1 and OriginState = 'CA';

<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>index_merge</td>
<td>OriginState,DepDel15</td>
<td>OriginState,DepDel15</td>
<td>3,5</td>
<td>NULL</td>
<td>76952</td>
<td>Using intersect(OriginState,DepDel15);Using where</td>
</tr>
</tbody>
</table>

# The select takes 2.20 sec

But it only works with equality conditions:

MySQL [ontime] > explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');

<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>ref</td>
<td>OriginState,DepDel15</td>
<td>DepDel15</td>
<td>5</td>
<td>const</td>
<td>36926</td>
<td>Using where</td>
</tr>
</tbody>
</table>

# The select takes 10.78 sec

MariaDB 5.3 doesn't have this limitation:

MariaDB [ontime] > explain select avg(arrdelay) from ontime where depdel15=1 and OriginState IN ('CA', 'GB');

<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>ontime</td>
<td>index_merge</td>
<td>OriginState,DepDel15,OriginState</td>
<td>DepDel15,OriginState</td>
<td>5,3</td>
<td>NULL</td>
<td>60754</td>
<td>Using sort_intersect(DepDel15,OriginState); Using where</td>
</tr>
</tbody>
</table>

# The select takes 3.23 sec
Intersect vs sort_intersect

- Different execution strategies

Regular intersect merges ordered streams

Sort-intersect doesn't require inputs to be ordered
set optimizer_switch='index_merge_sort_intersection=on';

explain
select max(l_extendedprice)
from lineitem
where
  l_partkey between 1 and 50000 and
  l_shipdate between '1993-07-01' and '1993-07-31';

*************************** 1. row ***************************
  id: 1
  select_type: SIMPLE
  table: lineitem
  type: index_merge
  possible_keys: i_l_shipdate,i_l_suppkey_partkey,i_l_partkey
    key: i_l_shipdate,i_l_partkey
    key_len: 4,5
    ref: NULL
    rows: 35823
  Extra: Using sort_intersect(i_l_shipdate,i_l_partkey); Using where
1 row in set (0.01 sec)
Benchmarking sort-intersect

Test runs
- hot/cold buffer pool,
- 4 queries with varying range sizes

Speedups:
- Cold: 7x
- Hot: 21x
New query optimizer features

- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- Batched Key Access
- Hash join
Batched Key Access

• Batched Key Access feature ("BKA")
  • Originally in (never released) MySQL 6.0
  • Now in MariaDB 5.3
    • With EXPLAIN showing more information
  • A number of bugs (the "MRR bug pile") were fixed
• New improvement “Key ordered retrieval”
Nested Loop Join
Nested loops join
Nested loops join

```
<table>
<thead>
<tr>
<th>table1</th>
<th>table2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Nested loops join

table1

table2
Nested loops join: outer loop
Nested loops join: lookups
Nested loops join: inner loops
BKA and Hash Join: background

Most of join execution time is spent here.
Batched Key Access idea
Why are batched index lookups faster?

- Non-BKA hits data at random
- That's hard on the cache
- Prefetching is not useful
Why are batched index lookups faster?

With BKA:

- We sort the buffer
- And then read data in order
- That's cache-friendly
- And prefetch-friendly

- Btw, MySQL 5.6's Multi-Range-Read is about reading in order. But only for range access
Batched Key Access parameters

- What if there is not enough space in the buffer?
- Multiple passes:

  BKA's most important parameter: @@join_buffer_size
BKA benchmark
set join_cache_level=6;

select max(l_extendedprice) 
  from orders, lineitem
where
  l_orderkey=o_orderkey and
  o_orderdate between $DATE1 and $DATE2

<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>
<th>rows</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>range</td>
<td>PRIMARY, i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>NULL</td>
<td>142680</td>
<td>Using where; Using index</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>lineitem</td>
<td>ref</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>orders.o_orderkey</td>
<td>2</td>
<td>Using join buffer (flat, BKA join); Key-ordered scan</td>
</tr>
</tbody>
</table>

Run with
• Various **join_buffer_size** settings
• Various size of $DATE1...$DATE2 range
New Batched Key Access Speedups

BKA join performance depending on buffer size

Performance without BKA

Performance with BKA, given sufficient buffer size

- Speedups: 4x-10x
Batched Key Access – how to use

- Documentation at [http://kb.askmonty.org](http://kb.askmonty.org)
- Google for “batched key access”

**Basic controls**
- `SET join_cache_level=6;`
- `SET join_buffer_size=10M..100M`
- `SET join_buffer_space_limit= ...`
  - per-query limit, i.e. `@@join_buffer_size * tables_in_join`
- `SET optimizer_switch='join_cache_hashed=off'`
  - (disable hash join)

- Then look at EXPLAINs:

<table>
<thead>
<tr>
<th>type</th>
<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>range</td>
<td>orders</td>
<td>range</td>
<td>i_o_orderdate</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>NULL</td>
<td>142680</td>
<td>Using where; Using index</td>
</tr>
<tr>
<td>ref</td>
<td>lineitem</td>
<td>ref</td>
<td>i_l_orderkey</td>
<td>i_l_orderkey</td>
<td>4</td>
<td>orders.o_orderkey</td>
<td>2</td>
<td>Using join buffer (flat, BKA join)</td>
</tr>
</tbody>
</table>
New query optimizer features

- Correct optimization of index_merge vs range
- index_merge/sort-intersect
- Batched Key Access
- Hash join
Hash join

- It's essentially “Using join buffer” with hashed access to the buffer
Targeted use cases

- There are no suitable indexes

- There is a suitable index

- but it's not sufficiently selective for BKA
### Hash Join Benchmark

MariaDB [dbt3sf10]> explain select max(l_discount) from part, lineitem
  where l_partkey = p_partkey and p_retailprice > 1800\G

```
 *************************** 1. row ***************************
  id: 1
  select_type: SIMPLE
  table: part
  type: ALL
  possible_keys: PRIMARY
    key: NULL
    key_len: NULL
    ref: NULL
    rows: 1991995
  Extra: Using where

 *************************** 2. row ***************************
  id: 1
  select_type: SIMPLE
  table: lineitem
    type: hash_ALL
  possible_keys: i_l_suppkey_partkey,i_l_partkey
    key: #hash#i_l_suppkey_partkey
    key_len: 5
    ref: dbt3sf10.part.p_partkey
    rows: 60133474
  Extra: Using join buffer (flat, BNLH join)
```

2 rows in set (0.00 sec)
Hash join benchmark

```sql
select max(l_discount)
from part, lineitem
where
    l_partkey = p_partkey and p_retailprice > 1800;
```

Speedups:
- 1x-5x wrt BKA
- Without BKA, the query didn't finish
Hash join summary

- “Classic” (i.e. not “GRACE”)
- Implemented as extension to BKA/join buffering scheme
- No full optimizer support, mostly switches/parameters so far.
- Execution part is finished and stable
- Optimization is more challenging
  - Hash join is OFF by default
  - Currently mostly controlled by server settings
- To try:
  - SET optimizer_switch='join_cache_hashed=on';
  - SET join_cache_level=4+;
  - SET join_buffer_size=...;

```
<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>orders</td>
<td>range</td>
<td>PRIMARY</td>
<td>i_o_orderdate</td>
<td>4</td>
<td>NULL</td>
<td>142680</td>
<td>Using where; Using index</td>
</tr>
<tr>
<td>lineitem</td>
<td>hash</td>
<td>PRIMARY</td>
<td>PRIMARY</td>
<td>4</td>
<td>orders.o_orderkey</td>
<td>60203402</td>
<td>Using join buffer (flat, BNLH join)</td>
</tr>
</tbody>
</table>
```
Conclusion

• New features
  • Correct optimization of index_merge vs range
  • index_merge/sort-intersect
  • Batched Key Access
  • Hash join

• All are/can be turned off unless we're sure the new feature can't make anything worse
  => you aren't “burning any bridges” when migrating to newer MariaDB.
Thanks

Q & A