Query Optimization with MySQL 5.6: Old and New Tricks

Jaime Crespo
Senior MySQL Instructor
Percona Live London 2013
November 11, 2013

About This Presentation

- Objective #1:
  - **Get faster queries**
- Reminder of some well-known query optimization techniques
  - Focus on new 5.6 query features
- For other performance tricks with MySQL & InnoDB:
  - *InnoDB Architecture and Performance Optimization* with Peter Zaitsev at 2pm

Agenda

1. Introduction and Environment Setup
2. Server Update and Configuration
3. Tools for Query Profiling
4. Indexing: Old and New Strategies
5. New Query Planner Features
6. Results and wrap-up

Example Platform

- Sample LAMP Application
  - 'My Movies' PHP Script
  - Example movie database
- Distributed as a VirtualBox VM
  - Import percona_live.ova
  - User: percona / pass: percona
  - Make sure httpd and PS are running
  - Optionally: startxfce4
- Currently no design optimization, no indexes (except PK), no proper query design

Server Update and Configuration

Basic Configuration Review

- InnoDB as the default engine (default in 5.5)
- InnoDB File Per Table (default in 5.6)
- Larger Buffer Pool and Transaction Log (changed in 5.6)
- Row Based Replication and tx_isolation
- Performance Schema overhead (enabled by default in 5.6)
- InnoDB page checksum (changed in 5.6)
- InnoDB Old Blocks time (changed in 5.6)

Query Optimization with MySQL 5.6: Old and New Tricks

TOOLS FOR QUERY PROFILING

EXPLAIN

- EXPLAIN is most important tool for the MySQL DBA and Developer
- It provides:
  - The query plan strategy
  - The index(es) to be used
  - The order of table access
  - An estimation of the number of rows to be examined

EXPLAIN on DML

- EXPLAIN is now available also for INSERT, UPDATE and DELETE

```
mysql> EXPLAIN DELETE FROM title WHERE title = 'Pilot'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: PRIMARY,title
          key: title
      key_len: 77
          ref: NULL
         rows: 1380
        Extra: Using where
1 row in set (0.00 sec)
```

Structured EXPLAIN


```
mysql> EXPLAIN FORMAT=JSON SELECT COUNT(*) FROM title
WHERE (title = 'Pilot' OR production_year > 2010)
AND kind_id < 4\G
*************************** 1. row ***************************
EXPLAIN: {
  "query_block": {
    "select_id": 1,
    "table": {
      "table_name": "title",
      "access_type": "index_merge",
      "possible_keys": [
        "title",
        "production_year"
      ],
      "key": "sort_union(title,production_year)",
      "key_length": "77,5",
      "rows": 4434,
      "filtered": 100,
      "attached_condition": "(((`imdb`.`title`.`title` = 'Pilot') or
(`imdb`.`title`.`production_year` > 2010)) and
(`imdb`.`title`.`kind_id` < 4))"
    }
  }
}
```

Optimizer Trace

- Allows profiling of MySQL query planner
- It shows not only information about the final query plan (EXPLAIN), but also about other discarded strategies, and its "execution cost"
- It can be accessed via the INFORMATION_SCHEMA database
- It is off by default

Checking the Optimizer Trace

- `mysql> SET optimizer_trace="enabled=on";`
- `mysql> SELECT COUNT(*) FROM title WHERE (title = 'Pilot' OR production_year > 2010) AND kind_id < 4;`

```
+-----------+
| COUNT(*)  |
+-----------+
|      3050 |
+-----------+
1 row in set (0.01 sec)
```

- `mysql> SELECT trace FROM information_schema.optimizer_trace;`

Checking the Optimizer Trace (cont.)


```
            {
              "range_scan_alternatives": [
                {
                  "index": "production_year",
                  "ranges": [
                    "2010 < production_year"
                  ],
                  "index_dives_for_eq_ranges": true,
                  "rowid_ordered": false,
                  "using_mrr": false,
                  "index_only": true,
                  "rows": 3054,
                  "cost": 615.16,
                  "chosen": true
                }
              ],
              "index_to_merge": "production_year",
              "cumulated_cost": 905.69
            }
          ],
          "cost_of_reading_ranges": 905.69,
          "cost_sort_rowid_and_read_disk": 3672.8,
          "cost_duplicate_removal": 9467.6,
          "total_cost": 14046
        }
      ],
      "chosen_range_access_summary": {
        "range_access_plan": {
          "type": "index_merge",
          "index_merge_of": [
```

Limitations of EXPLAIN

- It does not really execute the query
  - Not even subqueries any more
- The row statistics can be sometimes very off
- It does not give us information about temporary tables (on disk or memory?), subtask timing, query cache hit, etc.

Handler Statistics

- They provide time & memory status-independency:

```
mysql> SHOW SESSION STATUS LIKE 'Hand%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 1     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_external_lock      | 2     |
| Handler_mrr_init           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 0     |
| Handler_read_key           | 1     |
| Handler_read_last          | 0     |
| Handler_read_next          | 266   |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 0     |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 0     |
+----------------------------+-------+
18 rows in set (0.00 sec)
```

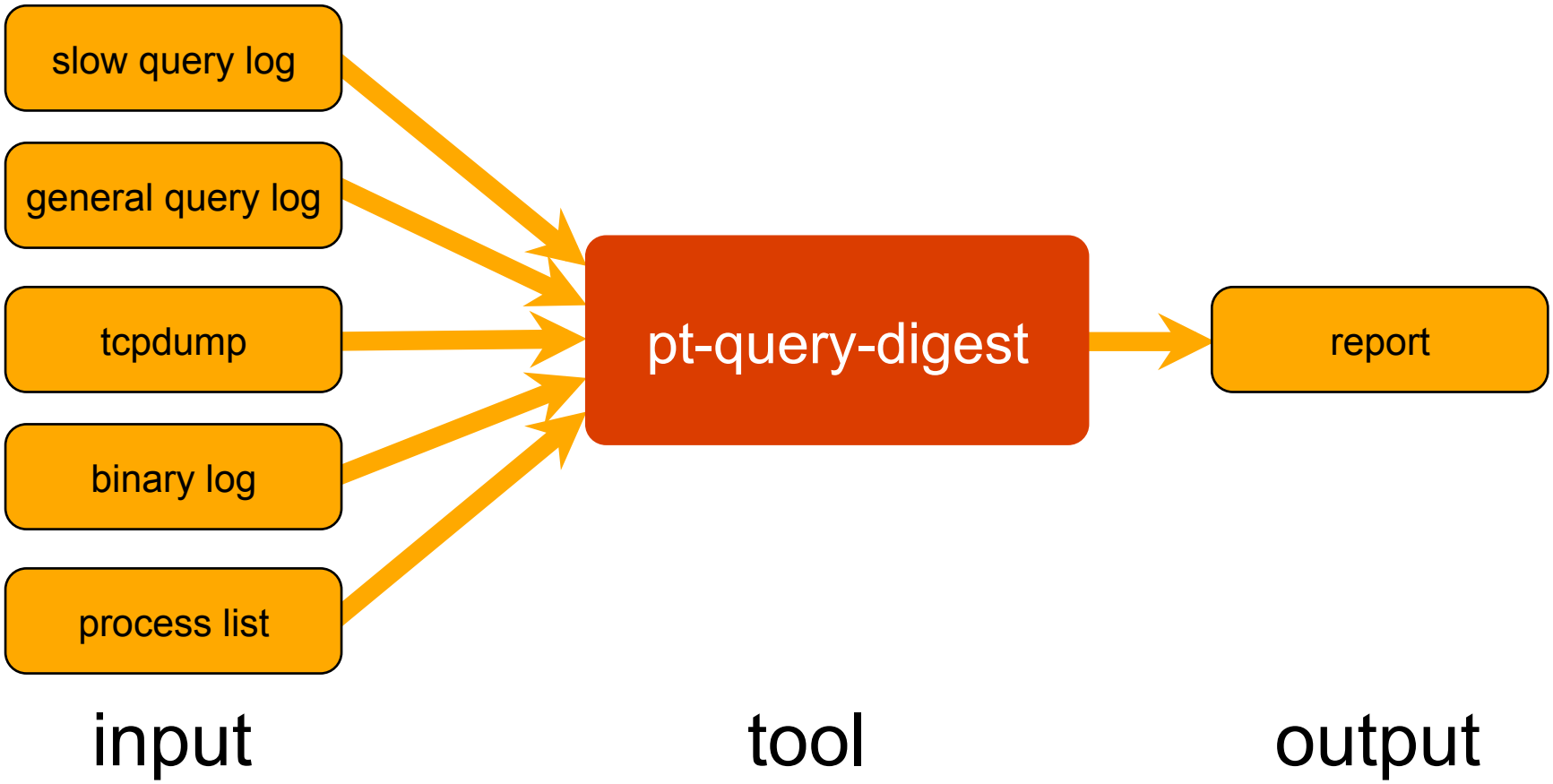
Slow Query Log and pt-query-digest

- Works on all MySQL versions
  - Specially useful in combination with the extended log statistics in Percona Server

```
SET GLOBAL log_slow_verbosity = 'full';
```

```
# Time: 130601  8:01:06.058915
# User@Host: root[root] @ localhost []  Id:    42
# Schema: imdb  Last_errno: 0  Killed: 0
# Query_time: 7.725616  Lock_time: 0.000328  Rows_sent: 4  Rows_examined: 1543720
Rows_affected: 0
# Bytes_sent: 272  Tmp_tables: 0  Tmp_disk_tables: 0  Tmp_table_sizes: 0
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: No  Tmp_table_on_disk: No
# Filesort: No  Filesort_on_disk: No  Merge_passes: 0
SET timestamp=1370073666;
SELECT id,title,production_year FROM title WHERE title = 'Bambi';
```

pt-query-digest



pt-query-digest Report

```
# Profile
# Rank Query ID           Response time   Calls R/Call  Apdx V/M   Item
# ==== ================== =============== ===== ======= ==== ===== =======
#    1 0xA8D2BBDE7EBE7822 4932.2992 28.8%    78 63.2346 0.00  5.22 SELECT person_info
#    2 0xFE25DAF5DBB71F49 4205.2160 24.6%   130 32.3478 0.00  3.47 SELECT title
#    3 0x70DAC639802CA233 1299.6269  7.6%    14 92.8305 0.00  0.17 SELECT cast_info
#    4 0xE336B880F4FEC4B8 1184.5101  6.9%   294  4.0289 0.36  2.29 SELECT cast_info
#    5 0x60550B93960F1837  905.1648  5.3%    60 15.0861 0.05  1.33 SELECT name
#    6 0xF46D5C09B4E0CA2F  777.2446  4.5% 16340  0.0476 1.00  0.17 SELECT char_name
#    7 0x09FCFFF0E5BC929F  747.4346  4.4%   130  5.7495 0.53  7.69 SELECT name
#    8 0x9433950BE12B9470  744.1755  4.4% 14368  0.0518 1.00  0.18 SELECT name
#    9 0x4DC0E044996DA715  448.5637  2.6%   130  3.4505 0.65  8.31 SELECT title
#   10 0x09FB72D72ED18E93  361.1904  2.1%    78  4.6306 0.28  1.89 SELECT cast_info title
#
# Query 1: 0.02 QPS, 1.40x concurrency, ID 0xA8D2BBDE7EBE7822 at byte 25744265
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count          0      78
# Exec time     28    4932s     11s    109s     63s     88s     18s     63s
# Lock time      0    724ms    36us   642ms     9ms    11ms    72ms    93us
# Rows sent      7   12.95k      86     334  170.05  329.68   89.24  136.99
# Rows examine  14 168.99M   2.17M   2.17M   2.17M   2.17M       0   2.17M
# Rows affecte   0       0       0       0       0       0       0       0
# Rows read      7   12.95k      86     334  170.05  329.68   89.24  136.99
# Bytes sent     5    2.39M  16.18k  54.14k  31.43k  54.03k  13.47k  25.99k
```

performance_schema

- Integrated profiling tool
  - Not new, but improved significately for 5.6
  - Substitutes the old PROFILING interface
  - Can have a 5-10% overhead by default
- Can be used to monitor several runtime statistics, among them, query execution statistics of the latest queries:

```
SELECT * FROM
performance_schema.events_statements_history_long
```

Query Optimization with MySQL 5.6: Old and New Tricks

Indexing: Old and New Strategies

Advantages of Indexing

- Faster Filtering
- Faster Column Return (Covering Index)
- Faster Ordering
- Certain Faster Functions (max, min)

Faster Filtering

- Indexes allow to return results by reading a fewer number of records
  - We should minimize the rows value in EXPLAIN/Handlers
  - `const > ref > range > index > ALL`
- Usually, only a single index can be used per table access
  - Compound indexes may provide higher filtering rate

Compound Indexes

- **Column Order in Index Does Matter**:
  - Order by decreasing selectivity
  - Only one range access per index*

Covering Index Technique

- Additional columns can be indexed in order to speed-up column retrieval
  - Specially interesting in n-to-n relationship tables
  - Not possible if the number or size of columns is too large (e.g. SELECT *, BLOBs)

Index Condition Pushdown

- Let's prepare a use case:

```
UPDATE cast_info SET note = left(note,
250);
ALTER TABLE cast_info MODIFY note
varchar(250), ADD INDEX role_id_note
(role_id, note);
```

- We want to execute:

```
SELECT * FROM cast_info
WHERE role_id = 1
and note like '%Jaime%';
```

Without ICP (5.5)

```
mysql> EXPLAIN SELECT *
FROM cast_info
WHERE role_id = 1
and note like '%Jaime%'\G
********** 1. row **********
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: role_id_note
          key: role_id_note
      key_len: 4
          ref: const
         rows: 11553718
        Extra: Using where
1 row in set (0.01 sec)
```

```
mysql> SHOW STATUS like 'Hand
   %';
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_commit             | 1       |
| Handler_delete             | 0       |
| Handler_discover           | 0       |
| Handler_prepare            | 0       |
| Handler_read_first         | 0       |
| Handler_read_key           | 1       |
| Handler_read_last          | 0       |
| Handler_read_next          | 8346769 |
| Handler_read_prev          | 0       |
| Handler_read_rnd           | 0       |
| Handler_read_rnd_next      | 0       |
| Handler_rollback           | 0       |
| Handler_savepoint          | 0       |
| Handler_savepoint_rollback | 0       |
| Handler_update             | 0       |
| Handler_write              | 0       |
+----------------------------+---------+
16 rows in set (0.00 sec)
```

With ICP (5.6)

```
mysql> EXPLAIN SELECT *
FROM cast_info
WHERE role_id = 1 and note like
'%Jaime%'\G
************ 1. row ************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: role_id_note
          key: role_id_note
      key_len: 4
          ref: const
         rows: 10259274
        Extra: Using index
condition
1 row in set (0.00 sec)
```

```
mysql> SHOW STATUS like 'Hand%';
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| Handler_commit             | 1     |
| Handler_delete             | 0     |
| Handler_discover           | 0     |
| Handler_external_lock      | 2     |
| Handler_mrr_init           | 0     |
| Handler_prepare            | 0     |
| Handler_read_first         | 0     |
| Handler_read_key           | 1     |
| Handler_read_last          | 0     |
| Handler_read_next          | 266   |
| Handler_read_prev          | 0     |
| Handler_read_rnd           | 0     |
| Handler_read_rnd_next      | 0     |
| Handler_rollback           | 0     |
| Handler_savepoint          | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update             | 0     |
| Handler_write              | 0     |
+----------------------------+-------+
18 rows in set (0.00 sec)
```

Comparison of ICP Execution

- Execution time for this example:
  - MySQL 5.5: 5.76 sec
  - MySQL 5.6: 1.09 sec
- Over 5x faster
- In this example, it would not work with a prefix index (e.g. TEXT/BLOB), as it must search the whole field
- Fun fact: if covering technique is tried, it actually runs slower than with ICP for this case/hardware (#68554)

ICP and Indexing

- ICP will change the way we index our tables

```
SELECT * FROM cast_info
FORCE INDEX(person_role_id_role_id)
WHERE person_role_id > 0
and person_role_id < 150000
and role_id > 1 and role_id < 7;
```

- For example, a multiple column index can be, in some cases, efficiently used for a range condition on several columns:
  - Effectiveness is highly dependent on how selective is the second part of the index (an “index scan” is still done at engine level)

Index for Sorts

- Additional columns can be indexed in order to speed up ordering of rows
  - Only effective in simple patterns
- GROUP BY operations are essentially sorts
  - They can also be greatly improved with the right indexes in some cases

Multi-Range Read (MRR)

- Reorders access to table data when using a secondary index on disk for sequential I/O
  - Like ICP, its efficiency is highly dependent on data distribution and memory contents
- It also depends on hardware sequential access (e.g., InnoDB's read ahead)
  - May not be useful on SSDs
- Not compatible with Using index
- Only for range access and equi-joins

MRR Example

- We want to execute:

```
SELECT * FROM cast_info
WHERE person_role_id > 0
and person_role_id < 150000;
SELECT * FROM cast_info
WHERE role_id = 3
and person_role_id > 0
and person_role_id < 500000;
```

- We will use these indexes, respectively:

```
ALTER TABLE cast_info
ADD INDEX person_role_id (person_role_id);
ALTER TABLE cast_info
ADD INDEX person_role_id_role_id (person_role_id,
role_id);
```

Without MRR (5.5)

```
mysql> EXPLAIN SELECT * FROM
cast_info FORCE
INDEX(person_role_id)  WHERE
person_role_id > 0 and
person_role_id < 150000\G
*********** 1. row ***********
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: range
possible_keys: person_role_id
          key: person_role_id
      key_len: 5
          ref: NULL
         rows: 8966490
        Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> SHOW STATUS like 'Hand%';
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_commit             | 1       |
| Handler_delete             | 0       |
| Handler_discover           | 0       |
| Handler_prepare            | 0       |
| Handler_read_first         | 0       |
| Handler_read_key           | 1       |
| Handler_read_last          | 0       |
| Handler_read_next          | 4654312 |
| Handler_read_prev          | 0       |
| Handler_read_rnd           | 0       |
| Handler_read_rnd_next      | 0       |
| Handler_rollback           | 0       |
| Handler_savepoint          | 0       |
| Handler_savepoint_rollback | 0       |
| Handler_update             | 0       |
| Handler_write              | 0       |
+----------------------------+---------+
16 rows in set (0.00 sec)
```

With MRR (5.6)

```
mysql> EXPLAIN SELECT * FROM
cast_info FORCE
INDEX(person_role_id)  WHERE
person_role_id > 0 and
person_role_id < 150000\G
*********** 1. row ***********
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: range
possible_keys: person_role_id
          key: person_role_id
      key_len: 5
          ref: NULL
         rows: 8966490
        Extra: Using index
condition; Using MRR
1 row in set (0.00 sec)
```

```
mysql> SHOW STATUS like 'Hand%';
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_commit             | 1       |
| Handler_delete             | 0       |
| Handler_discover           | 0       |
| Handler_external_lock      | 4       |
| Handler_mrr_init           | 0       |
| Handler_prepare            | 0       |
| Handler_read_first         | 0       |
| Handler_read_key           | 4654313 |
| Handler_read_last          | 0       |
| Handler_read_next          | 4654312 |
| Handler_read_prev          | 0       |
| Handler_read_rnd           | 4654312 |
| Handler_read_rnd_next      | 0       |
| Handler_rollback           | 0       |
| Handler_savepoint          | 0       |
| Handler_savepoint_rollback | 0       |
| Handler_update             | 0       |
| Handler_write              | 0       |
+----------------------------+---------+
18 rows in set (0.00 sec)
```

Comparison of MRR Execution

- Execution time for this example:
  - MySQL 5.5: 4654312 rows in set (1 min 4.79 sec)
  - MySQL 5.6 (w/MRR, wo/ICP): 4654312 rows in set (48.64 sec)
- Consistent 33% execution improvement for this test case
  - Difficult to see for smaller resultsets
- What has changed?
  - 15% more “read_ahead”s, resulting in a 40% less data reads
  - Reordering has an overhead, can be tuned with read_rnd_buffer_size

What's the Point for the 'FORCE INDEX'?

- For demonstration purposes (full table scan is better here)
- Some features are not properly detected yet:

```
SELECT *
FROM cast_info FORCE INDEX(person_role_id_role_id)
WHERE role_id = 3
and person_role_id > 0
and person_role_id < 500000;
```

- Full table scan (any version): Empty set (8.08 sec)
- MySQL 5.5: Empty set (1 min 16.88 sec)
- MySQL 5.6 ICP+MRR*: Empty set (0.46 sec)

*ICP is responsible for the dramatic change in execution time, not MRR

Batched Key Access (BKA)

- It retrieves keys in batches and allows MRR usage for JOINs, as an alternative to standard Nested Loop Join execution
- Not enabled by default

```
SET optimizer_switch=
'mrr=on,mrr_cost_based=off,batched_key_access=on';
```

Without BKA (5.5) - EXPLAIN

```
mysql> EXPLAIN
SELECT cast_info.note, char_name.name
FROM cast_info FORCE index(person_role_id)
JOIN char_name
ON cast_info.person_role_id = char_name.id
WHERE cast_info.person_role_id > 0 and
cast_info.person_role_id < 150000
************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 313782
        Extra: Using where
...
```

```
...
************* 2. row *************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id
          key: person_role_id
      key_len: 5
          ref: imdb.char_name.id
         rows: 4
        Extra: NULL
2 rows in set (0.00 sec)
```

Without BKA (5.5) - Handlers

```
mysql> SHOW STATUS like 'Hand%';
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_commit             | 1       |
| Handler_delete             | 0       |
| Handler_discover           | 0       |
| Handler_prepare            | 0       |
| Handler_read_first         | 0       |
| Handler_read_key           | 150000  |
| Handler_read_last          | 0       |
| Handler_read_next          | 4804311 |
| Handler_read_prev          | 0       |
| Handler_read_rnd           | 0       |
| Handler_read_rnd_next      | 0       |
| Handler_rollback           | 0       |
| Handler_savepoint          | 0       |
| Handler_savepoint_rollback | 0       |
| Handler_update             | 0       |
| Handler_write              | 0       |
+----------------------------+---------+
18 rows in set (0.00 sec)
```

With BKA (5.6) - EXPLAIN

```
mysql> EXPLAIN
SELECT cast_info.note, char_name.name
FROM cast_info FORCE index(person_role_id)
JOIN char_name ON cast_info.person_role_id
= char_name.id
WHERE cast_info.person_role_id > 0 and
cast_info.person_role_id < 150000\G
************ 1. row ************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 313782
        Extra: Using where
...
```

```
...
************ 2. row ************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id
          key: person_role_id
      key_len: 5
          ref: imdb.char_name.id
         rows: 4
        Extra: Using join buffer
(Batched Key Access)
2 rows in set (0.00 sec)
```

With BKA (5.6) - Handlers

```
mysql> SHOW STATUS like 'Hand%';
+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| Handler_commit             | 1       |
| Handler_delete             | 0       |
| Handler_discover           | 0       |
| Handler_external_lock      | 6       |
| Handler_mrr_init           | 1       |
| Handler_prepare            | 0       |
| Handler_read_first         | 0       |
| Handler_read_key           | 4804312 |
| Handler_read_last          | 0       |
| Handler_read_next          | 4804311 |
| Handler_read_prev          | 0       |
| Handler_read_rnd           | 4654312 |
| Handler_read_rnd_next      | 0       |
| Handler_rollback           | 0       |
| Handler_savepoint          | 0       |
| Handler_savepoint_rollback | 0       |
| Handler_update             | 0       |
| Handler_write              | 0       |
+----------------------------+---------+
18 rows in set (0.00 sec)
```

Comparison of BKA Execution

- Execution time for this example:
  - MySQL 5.5: 4654312 rows in set (1 min 6.78 sec)
  - MySQL 5.6 (w/MRR, w/BKA): 4654312 rows in set (1 min 0.47 sec)
- The results are consistent between executions, but the gain is not too big. But if we change the join_buffer_size…
  - MySQL 5.6 (w/MRR, w/BKA, join_buffer_size = 50M): 4654312 rows in set (19.54 sec)
    (join_buffer_size does not affect execution time in the 5.5 version)

Better Index Merge

- In this example, it avoids a full table scan:

```
mysql>  EXPLAIN SELECT COUNT(*) FROM title WHERE (title =
'Pilot' OR production_year > 2010) AND kind_id < 4\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: index_merge
possible_keys: title,production_year
          key: title,production_year
      key_len: 77,5
          ref: NULL
         rows: 4434
        Extra: Using sort_union(title,production_year);
Using where
1 row in set (0.00 sec)
```

(almost all titles have kind_id < 4)

- MySQL 5.5: 0.79s – MySQL 5.6: 0.01s

Extended Secondary Keys

- Implicit primary keys inside secondary keys can be used for filtering (ref, range, etc), not only for covering index or sorting.
- Requires `use_index_extensions=on` (default)

```
ALTER TABLE title add index
(title(25));
SELECT COUNT(*) FROM title
WHERE title = 'Pilot'
AND id BETWEEN 1000 AND 1000000;
```

Extended Secondary Keys

```
mysql> EXPLAIN SELECT COUNT(*) FROM title WHERE
title = 'Pilot' AND id BETWEEN 1000 AND
1000000\G
******************** 1. row ********************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: PRIMARY,title
          key: title
      key_len: 81
          ref: NULL
         rows: 531
        Extra: Using index condition; Using
where
1 row in set (0.00 sec)
```

Duplicate Key Check


```
mysql> alter table title add index (production_year);
Query OK, 0 rows affected (10.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql> alter table title add index (production_year);
Query OK, 0 rows affected, 1 warning (5.11 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> show warnings\G
*************************** 1. row ***************************
  Level: Note
   Code: 1831
Message: Duplicate index 'production_year_2' defined on
the table 'imdb.title'. This is deprecated and will be
disallowed in a future release.
1 row in set (0.00 sec)
```

pt-duplicate-key-checker

- It detects duplicate and redundant indexes
  - Including primary keys at the end of secondary keys on InnoDB tables

pt-index-usage & user_stats

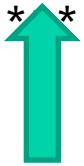
- Can detect unused indexes
  - That slow down writes
  - Can also affect reads!
    - Wasted memory
    - Query optimizer woes

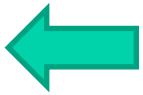
New Query Planner Features

Filesort with Short LIMIT

- Queries that require a filesort but only the first records are selected can benefit from this optimization:

```
mysql> EXPLAIN select * from movie_info
ORDER BY info LIMIT 100\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: movie_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6927988
        Extra: Using filesort
1 row in set (0.00 sec)
```





Note: Both EXPLAIN and the STATUS Handlers show the same outcome

Filesort with Short LIMIT (cont.)

- `SELECT * FROM movie_info ORDER BY info LIMIT 100;`
  - MySQL 5.5: 20.06 sec
  - MySQL 5.6 (P_S on): 9.14 sec
  - MySQL 5.6 (P_S off): 8.51 sec
- Over 2x faster.
  - Exact speed-up may depend on the original sort buffer size

Subquery Optimization

- Late Subquery Materialization
  - Useful for FROM subqueries if further filtering is done
  - MySQL may also decide to create its own index on a temporary table on memory

```
SELECT * FROM title JOIN (SELECT * FROM cast_info)
AS subselect ON subselect.movie_id = title.id WHERE
subselect.id = 1;
```

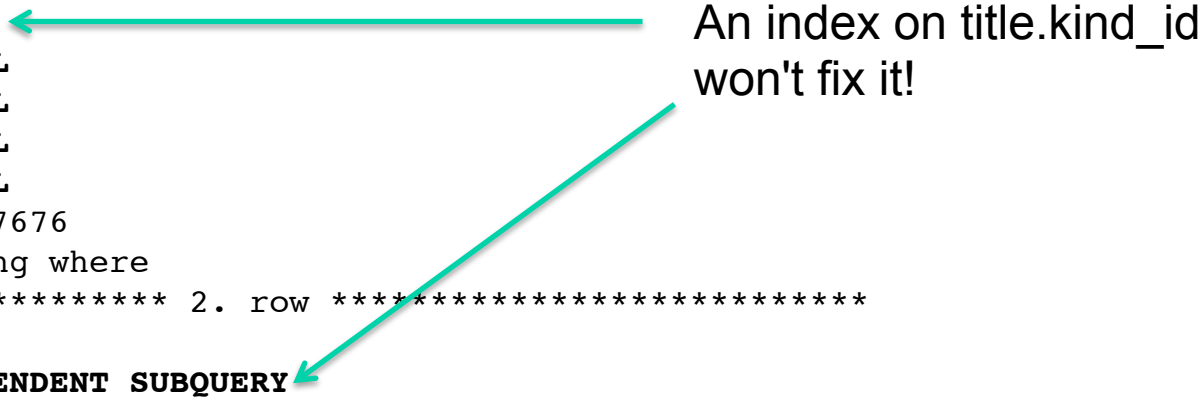
EXPLAIN: 0.00 sec (vs >2 min in 5.5)
EXECUTION: 1m 22 sec (vs > 4 min in 5.5)

JOINs and Subqueries

- Better detection of IN + non-dependent subqueries
  - Many queries do not need to be converted to a JOIN anymore for a proper execution plan
  - MySQL decides what is the best strategy: Materialization, Semi-Join, etc.

That Subquery in 5.5

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1567676
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)
```



An index on title.kind_id won't fix it!

That Subquery in 5.6

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN (SELECT id FROM kind_type WHERE
kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: kind_type
         type: const
possible_keys: PRIMARY,kind
          key: kind
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1477989
        Extra: Using where
2 rows in set (0.00 sec)
```

An index on title.kind_id
can and will be very useful!

Join Order

- Table order algorithm has been optimized, which leads to better query plans when joining many tables

Persistent Optimizer Statistics

- InnoDB index statistics are no longer discarded on server shutdown and recomputed the next time a table is accessed
- Controlled by variable:
  `innodb_stats_persistent = ON` (default)
- Remember that SHOW commands/accessing to I_S do not automatically regenerate statistics by default

READ ONLY Transactions

- START TRANSACTION READ ONLY;
  - It avoids some overhead of R/W transactions (MVCC)
  - Automatically used for SELECT in auto_commit
  - Once started, a new transaction has to be started to perform a write

Explicit Partition Selection

- Usually unnecessary, as MySQL does automatically partition pruning
- Can be useful combined with some non-deterministic functions or in some special cases:

```
SELECT count(*)
    FROM title PARTITION(p01, p02);
```

Query Optimization with MySQL 5.6: Old and New Tricks

Results and Wrap-Up

5.6 is a Great Release

- Many optimizer changes that can potentially improve query execution time
- Some of them are transparent, some others require tuning and understanding
  - Some old tricks and indexing strategies become obsolete in 5.6
- pt-upgrade, from Percona Toolkit, and Percona Playback can be great tools to analyze improvements and regressions

Where to Learn More ?

- More query and server optimization techniques in our training courses (http://www.percona.com/training):
  - Scaling and Optimization for MySQL
  - Operations and Troubleshooting with MySQL
  - Moving to MySQL 5.6
  - Analysing Queries with Percona Toolkit
  - Percona Server and Tools (NEW!)
- Ask me for a discount (only to PLUK attendants)!

Where to Learn More ? (cont.)

- Books:
  - High Performance MySQL, 3rd Edition
  - SQL Antipatterns
- Free MySQL Webinars:
  - http://www.percona.com/resources/mysql-webinars

PERCONA

Percona Live MySQL Conference and Expo 2014

Learn from Leading MySQL Experts
Santa Clara, CA, April 1st – 4th, 2013

- **For more information and to register:**
  Visit: http://www.percona.com/live/mysql-conference-2014/

Thank You!

Jaime Crespo
jaime.crespo@percona.com