MySQL Query Optimization

Kenny Gryp <kenny.gryp@percona.com>
Percona Live Washington DC / 2012-01-11
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
MySQL Query Optimization

• The number one goal is to have faster queries.
• The process is:
  • We first ask MySQL what its intended execution plan is.
  • If we don't like it, we make a change, and try again...
• More Information:
  • High Performance MySQL, 2nd Edition:
    http://shop.oreilly.com/product/9780596101718.do
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
First Example

1. CREATE TABLE `title` (  
2.   `id` int(11) NOT NULL AUTO_INCREMENT,  
3.   `title` text NOT NULL,  
4.   `imdb_index` varchar(12) DEFAULT NULL,  
5.   `kind_id` int(11) NOT NULL,  
6.   `production_year` int(11) DEFAULT NULL,  
7.   `imdb_id` int(11) DEFAULT NULL,  
8.   `phonetic_code` varchar(5) DEFAULT NULL,  
9.   `episode_of_id` int(11) DEFAULT NULL,  
10.  `season_nr` int(11) DEFAULT NULL,  
11.  `episode_nr` int(11) DEFAULT NULL,  
12.  `series_years` varchar(49) DEFAULT NULL,  
13.  PRIMARY KEY (`id`)  
14. ) ENGINE=InnoDB AUTO_INCREMENT=1543721;
Find the Title Bambi

1. mysql> EXPLAIN SELECT id, title, production_year FROM title
2. WHERE title = 'Bambi' ORDER BY production_year\G
3. ******************************************************************************
4.  id: 1
5.  select_type: SIMPLE
6.   table: title
7.     type: ALL
8.  possible_keys: NULL
9.  key: NULL
10. key_len: NULL
11. ref: NULL
12. rows: 1900087
13. Extra: Using where; Using filesort;
14. 1 row in set (0.00 sec)
Find the Title Bambi

1. mysql> EXPLAIN SELECT id, title, production_year FROM title
2. WHERE title = 'Bambi' ORDER BY production_year
3. ****************************** 1. row ******************************
4. id: 1
5. select_type: SIMPLE
6. table: title
7. type: ALL
8. possible_keys: NULL
9. key: NULL
10. key_len: NULL
11. ref: NULL
12. rows: 1900087
13. Extra: Using where; Using filesort;
14. 1 row in set (0.00 sec)

ALL means tablescan

In this case a sort is required because of the order by, but not for all rows, only matching rows

Additional filtering may be possible before passing to sort.
Specify Index Length

1. `mysql> ALTER TABLE title ADD INDEX (title);`
2. `ERROR 1170 (42000): BLOB/TEXT column 'title' used in key specification without a key length`
3. 
4. 
5. 
6. `mysql> ALTER TABLE title ADD INDEX (title(50));`
7. `Query OK, 1543719 rows affected (1 min 19.14 sec)`
8. `Records: 1543719  Duplicates: 0  Warnings: 0`
Size of Key?

- Length of index is limited:
  - 1000 bytes MyISAM
  - 767 bytes InnoDB
- UTF-8 uses up to 3 bytes (3 bytes are used in determining)
- Variable length strings (VARCHAR, TEXT...): add 1 or 2 bytes for length
We must revisit...

```
1. mysql> EXPLAIN SELECT id, title, production_year FROM title
2. WHERE title = 'Bambi' ORDER by production_year\G
3. *************************** 1. row ***************************
4.     id: 1
5.     select_type: SIMPLE
6.       table: title
7.         type: ref
8.     possible_keys: title
9.         key: title
10.     key_len: 152
11.     ref: const
12.     rows: 4
13.     Extra: Using where; Using filesort;
14.  1 row in set (0.14 sec)
```

Much Faster!
We must revisit...

Using = for comparison, but not primary key lookup.

Identified title as a candidate index, chose to use it.

Size of the index used (in bytes)

Anticipated number of rows to be examined dropped considerably.
Other ways of accessing

```sql
mysql> EXPLAIN SELECT id,title,production_year FROM title
WHERE id = 55327\G
1. row *************************************************
4. id: 1
5. select_type: SIMPLE
6. table: title
7. type: const
8. possible_keys: PRIMARY
9. key: PRIMARY
10. key_len: 4
11. ref: const
12. rows: 1
13. Extra:
14. 1 row in set (0.06 sec)
```

At most one matching row.

In InnoDB the primary key is often much faster than all other keys.
Range Scan

Type is range. BETWEEN, IN() and < > are also ranges.

Anticipated number of rows to be examined has increased - we are not specific enough.

Ignore the time with EXPLAIN. Only look at the time for a query.
Why’s that a range?

- We're looking for titles between Bamb\text{A} and Bamb\text{Z}*
- When we say index in MySQL, we mean trees.
  - That is, B-Tree/B+Tree/T-Tree.
  - Pretend they're all the same (for simplification)
  - There's no radically different indexing methods in MySQL unless you play storage engine Bingo**.

* In reality the range is a little wider
** The memory & ndb storage engine supports hash indexes
What’s that?
Could this be a range?

```
1. mysql> EXPLAIN SELECT id, title, production_year FROM title
2. WHERE title LIKE '%ulp Fiction'\G
3. *************************** 1. row ***************************
4.   id: 1
5.   select_type: SIMPLE
6.     table: title
7.     type: ALL
8.   possible_keys: NULL
9.     key: NULL
10.    key_len: NULL
11.     ref: NULL
12.    rows: 1900087
13.   Extra: Using where;
14.  1 row in set (0.00 sec)
```
No, we can’t traverse.

Do we head left or right here?
LIKE 'Z%'
LIKE ‘T%’

1. mysql> EXPLAIN SELECT id,title,production_year FROM title
2. WHERE title LIKE 'T%'
3. **************************** 1. row ****************************
4.     id: 1
5.     select_type: SIMPLE
6.     table: title
7.     type: ALL
8.     possible_keys: title
9.     key: NULL
10.    key_len: NULL
11.    ref: NULL
12.    rows: 1492490
13.    Extra: Using where
14. 1 row in set (0.00 sec)
LIKE ‘The %’

1. mysql> EXPLAIN SELECT id,title,production_year FROM title
2. WHERE title LIKE 'The %'
3. **************************** 1. row ****************************
4.     id: 1
5.     select_type: SIMPLE
6.     table: title
7.     type: ALL
8.     possible_keys: title
9.     key: NULL
10.    key_len: NULL
11.    ref: NULL
12.    rows: 1492490
13.    Extra: Using where
14.    1 row in set (0.00 sec)
MySQL is (reasonably) smart.

- It dynamically samples the data to choose which is the better choice - or in some cases uses static statistics*.

- This helps the optimizer choose:
  - Which indexes will be useful.
  - Which indexes should be avoided.
  - Which is the better index when there is more than one.

* To refresh statistics run `ANALYZE TABLE table_name;`
Why avoid indexes?

- B-Trees work like humans search a phone book;
  - Use an index if you want just a few rows.
  - Scan cover-to-cover if you want a large percentage.
Why avoid indexes (cont.)

- We benchmarked this on a different schema:

Table scan has a relatively fixed cost (red line).

The index has completely different effectiveness depending on how much it can filter.

Hopefully MySQL switches at the right point.
What you should take away:

- **Data is absolutely critical.**
  - Development environments should contain sample data exported from production systems.

- **Input values are absolutely critical.**
  - Between two seemingly identical queries, execution plans may be *very* different.

Anticipated Number Of Rows

This number of rows is a guess. It keeps changing between examples.
Statistics Sampling

- InnoDB only keeps statistics samples in memory - and not on disk*.

- Sampling is performed when a table is first opened, and estimates are based on an estimate from sampling 8 random pages.

  - This number is used whether the table have 10 rows or 10 million rows.

In InnoDB plugin this is now configurable with `innodb_stats_sample_pages`. The setting is global, and will apply to all tables.

* In XtraDB 12 statistics can be retained with `innodb_use_sys_stats_table`=1.
* MySQL 5.6: `innodb_stats_persistent_sample_pages`
Statistics (cont.)

• Statistics are automatically regenerated on most meta-data commands:
  • SHOW TABLE STATUS
  • SHOW INDEX
  • Information Schema commands.

• As well as:
  • When the table size changes by more than 1/16th.
  • If more than 2,000,000,000 rows have been inserted.

Disable with `innodb_stats_on_metadata=0` (5.1 and above).

Disable with `innodb_stats_auto_update=0` (XtraDB only).
Improve this Query

1. mysql> ALTER TABLE title DROP INDEX title;
2. mysql> EXPLAIN SELECT * FROM title WHERE title = 'Pilot' AND production_year BETWEEN 1997 and 2009\G
3. **************************** 1. row ****************************
4.    id: 1
5.  select_type: SIMPLE
6.    table: title
7.  type: ALL
8.  possible_keys: NULL
9.  key: NULL
10. key_len: NULL
11. ref: NULL
12. rows: 1569823
13. Extra: Using where
14. 1 row in set (0.00 sec)
We’re Spoiled for Choice.

1. # Which one is best?
2. # ALTER TABLE title ADD INDEX py (production_year);
3. # ALTER TABLE title ADD INDEX t (title);
4. # ALTER TABLE title ADD INDEX py_t (production_year, title);
5. # ALTER TABLE title ADD INDEX t_py (title, production_year);
6. 
7. # Start by trying the production_year example:
8. mysql> ALTER TABLE title ADD INDEX py (production_year);
9. Query OK, 1543719 rows affected (38.07 sec)
10. Records: 1543719  Duplicates: 0  Warnings: 0
Index on production_year

1. mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
2. AND production_year BETWEEN 1997 and 2009\G
3. *********************************
4. 1. row ******************************************************
5. id: 1
6. select_type: SIMPLE
7. table: title
8. type: ALL
9. possible_keys: py
10. key: NULL
11. key_len: NULL
12. ref: NULL
13. rows: 1592559
14. Extra: Using where
15. 1 row in set (0.02 sec)
Might work if...

1. mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
2. AND production_year BETWEEN 2006 and 2009\G
3. ************************************************** 1. row **************************************************
4.      id: 1
5.      select_type: SIMPLE
6.        table: title
7.        type: range
8.      possible_keys: py
9.        key: py
10.     key_len: 5
11.     ref: NULL
12.     rows: 148320
13.     Extra: Using where
14.     1 row in set (0.00 sec)
Index on title(50)

```
1. mysql> ALTER TABLE title ADD INDEX t (title(50));
2. Query OK, 1543719 rows affected (38.07 sec)
3. Records: 1543719  Duplicates: 0  Warnings: 0
4. mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
5. AND production_year BETWEEN 2006 and 2009\G
6. ******************************************* 1. row *******************************************
7.     id: 1
8.     select_type: SIMPLE
9.     table: title
10.    type: ref
11.    possible_keys: py,t
12.     key: t
13.    key_len: 152
14.     ref: const
15.    rows: 926
16.    Extra: Using where
17.    1 row in set (0.00 sec)
```
Comparing the two:

```
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot' AND production_year BETWEEN 2006 and 2009\G
```

```
<table>
<thead>
<tr>
<th></th>
<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>1</td>
<td>SIMPLE</td>
<td>title</td>
<td>range</td>
<td>py</td>
<td>py</td>
<td>5</td>
<td>NULL</td>
<td>148320</td>
<td>Using where</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>key_len</td>
<td></td>
<td>152</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ref</td>
<td></td>
<td>const</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rows</td>
<td></td>
<td>926</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Extra</td>
<td></td>
<td>Using where</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 row in set (0.00 sec)</td>
</tr>
</tbody>
</table>
```
Composite Indexes.

• What is better?
  • INDEX py_t (production_year, title)
  • INDEX t_py (title, production_year)
Index on py_t

1. `mysql> ALTER TABLE title ADD INDEX py_t`
2. `(production_year, title(50));`
3. Query OK, 1543719 rows affected (2 min 15.64 sec)
4. Records: 1543719  Duplicates: 0  Warnings: 0
5.
6. `mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot' AND production_year BETWEEN 2006 and 2009\G`
7. `EXPLAIN SELECT * from title WHERE title = 'Pilot' AND production_year BETWEEN 2006 and 2009\G`
8. `id: 1`
9. `select_type: SIMPLE`
10. `table: title`
11. `type: ref`
12. `possible_keys: py,t,py_t`
13. `key: t`
14. `key_len: 152`
15. `ref: const`
16. `rows: 926`
17. `Extra: Using where`

Index on t_py

1. mysql> ALTER TABLE title ADD INDEX t_py
2. (title(50), production_year);
3. Query OK, 1543719 rows affected (1 min 52.63 sec)
4. Records: 1543719  Duplicates: 0  Warnings: 0
5. 
6. mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
7. AND production_year BETWEEN 2006 and 2009\G
8. ****************************** 1. row ******************************
9.   id: 1
10. select_type: SIMPLE
11.   table: title
12.     type: range
13. possible_keys: py,t,py_t,t_py
14.     key: t_py
15.     key_len: 157
16.     ref: NULL
17.     rows: 82
18.     Extra: Using where
Recommendations

• Index over multiple columns if it can improve filtering. i.e.
  • **GOOD:** Only some pilots made between 2006-2009.
  • **BAD:** All pilots made between 2006-2009.
Don't know what order to specify the columns?

**RULE:** Think how to filter the fastest. Use that order left to right.

**EXCEPTION:** If there's a range (>, BETWEEN, %). Those always go to the RIGHT.

- After a column is used for rangescan, groupby, you cannot use the next column in the composite index.

A quick sidetrack...

• So far indexes have only been used for filtering.
  • This is the most typical case - don’t forget it.

• There are also some other ways that MySQL can use an index
  • To avoid having to sort.
  • To prevent temporary tables.
  • To avoid reading rows.
  • ..
The first example again

```sql
1. mysql> EXPLAIN SELECT id, title, production_year FROM title
2. WHERE title = 'Bambi' ORDER BY production_year\G
3.  ********************************** 1. row **********************************
4.  id: 1
5.  select_type: SIMPLE
6.  table: title
7.  type: ALL
8.  possible_keys: NULL
9.  key: NULL
10. key_len: NULL
11. ref: NULL
12. rows: 1900087
13. Extra: Using where; Using filesort;
14. 1 row in set (0.00 sec)
```
Index prevents sort

1. mysql> ALTER TABLE title ADD INDEX t_py (title(50), production_year);
2. 
3. mysql> EXPLAIN SELECT id,title,production_year FROM title WHERE title = 'Bambi' ORDER BY production_year
4. ******************************************** 1. row ********************************************
5.     id: 1
6.     select_type: SIMPLE
7.     table: title
8.     type: ref
9.     possible_keys: t_py
10.    key: t_py
11.    key_len: 152
12.    ref: const
13.    rows: 4
14.    Extra: Using where
15. 1 row in set (0.00 sec)
Temporary Table

1. `mysql> EXPLAIN select count(*) as c, production_year FROM title`  
2. `GROUP BY production_year`  
3. `G`  
4. `1. row ******************************************`  
5. `id: 1`  
6. `select_type: SIMPLE`  
7. `table: title`  
8. `type: ALL`  
9. `possible_keys: NULL`  
10. `key: NULL`  
11. `key_len: NULL`  
12. `ref: NULL`  
13. `rows: 1524577`  
14. `Extra: Using temporary; Using filesort`  
15. `1 row in set (0.00 sec)`
“Loose index scan”

1. mysql> EXPLAIN select count(*) as c, production_year FROM title
2. GROUP BY production_year\G
3. ************************************** 1. row **************************************
4. id: 1
5. select_type: SIMPLE
6. table: title
7. type: index
8. possible_keys: NULL
9. key: py
10. key_len: 5
11. ref: NULL
12. rows: 1524577
13. Extra: Using index
14. 1 row in set (0.00 sec)

ALTER TABLE title
ADD INDEX py
(production_year);
Retrieving only limited columns

Query:

```sql
SELECT person_id FROM cast_info WHERE person_role_id = 35721;
```
Retrieving only limited columns:

1. mysql> EXPLAIN SELECT person_id FROM cast_info
2.    WHERE person_role_id = 35721
3.  ************************************************** 1. row **************************************************
4.                  id: 1
5.  select_type: SIMPLE
6.    table: cast_info
7.    type: ref
8.  possible_keys: person_role_id
9.    key: person_role_id
10.   key_len: 5
11.    ref: const
12.   rows: 146
13.  Extra: Using where
14.  1 row in set (0.01 sec)

ALTER TABLE cast_info ADD INDEX (person_role_id);
Covering Index Optimization:

1. mysql> EXPLAIN SELECT person_id FROM cast_info
2.   WHERE person_role_id = 35721\G
3.  ***************************************************** 1. row *****************************************************
4.       id: 1
5.   select_type: SIMPLE
6.       table: cast_info
7.       type: ref
8.   possible_keys: person_role_id, person_role_id_person_id
9.       key: person_role_id_person_id
10.      key_len: 5
11.     ref: const
12.       rows: 146
13.     Extra: Using where; Using index
14.   1 row in set (0.00 sec)

ALTER TABLE cast_info ADD INDEX person_role_id_person_id(person_role_id,
Key_len in EXPLAIN

key_len: 152

- How much bytes of the index is being used for the query?
- In composite indexes, can be used to determine how many columns will be used to _filter_ on.
Key Length Example

1. mysql> EXPLAIN SELECT person_id FROM cast_info
2. WHERE person_role_id = 35721\G
3. ******************************************** 1. row ********************************************
4.        id: 1
5.    select_type: SIMPLE
6.        table: cast_info
7.        type: ref
8. possible_keys: person_role_id,person_role_id_person_id
9.        key: person_role_id_person_id
10.   key_len: 5
11.        ref: const
12.        rows: 146
13. Extra: Using where; Using index
14. 1 row in set (0.00 sec)

key_len is only for WHERE clause, not the covering part
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
The limitations of EXPLAIN

- `EXPLAIN` shows MySQL’s intentions - there is no post execution analysis.
  - How many rows actually had to be sorted?
  - Was that temporary table created on disk?
  - Did the `LIMIT 10` result in a quick match, resulting in fewer rows scanned?
  - .. we don’t know.
More Advanced

• Combine EXPLAIN with other MySQL diagnostics:
  • SHOW SESSION STATUS
    • Recommended to run before and after the query.
    • Available in MySQL 5.0+
  • SHOW PROFILES
    • Available in 5.0 (limited), 5.1.
    • Breaks down the time taken on various steps of query execution.
    • Huge amount of skew in any numbers it reports under Linux.
  • Slow Query Log Extended Statistics (Percona Server)
    • Will let you know examined rows, temp table on disk, sort on disk, how many IOPS in InnoDB etc.
mysql-5141> EXPLAIN select STRAIGHT_JOIN count(*) as c, person_id
FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON
(cast_info.movie_id=title.id) WHERE title.kind_id = 1
GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1

2 rows in set (0.00 sec)

MySQL says that only 8 rows were examined in 5.1.41

Find the actor that stared in the most movies.
mysql5089> EXPLAIN select STRAIGHT_JOIN count(*) as c, person_id
FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON
(cast_info.movie_id=title.id) WHERE title.kind_id = 1
GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1;

1. row
---
id: 1
select_type: SIMPLE
table: cast_info
type: index
possible_keys: NULL
key: person_id
key_len: 8
ref: NULL
rows: 22187768
Extra: Using index; Using temporary; Using filesort

2. row
---
id: 1
select_type: SIMPLE
table: title
type: eq_ref
possible_keys: PRIMARY,title_kind_id_exists
key: PRIMARY
key_len: 4
ref: imdb.cast_info.movie_id
rows: 1
Extra: Using where
2 rows in set (0.00 sec)
Double Checking

<table>
<thead>
<tr>
<th>Variable_name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handler_commit</td>
<td>0</td>
</tr>
<tr>
<td>Handler_delete</td>
<td>0</td>
</tr>
<tr>
<td>Handler_discover</td>
<td>0</td>
</tr>
<tr>
<td>Handler_prepare</td>
<td>0</td>
</tr>
<tr>
<td>Handler_read_first</td>
<td>1</td>
</tr>
<tr>
<td>Handler_read_key</td>
<td>13890229</td>
</tr>
<tr>
<td>Handler_read_next</td>
<td>14286456</td>
</tr>
<tr>
<td>Handler_read_prev</td>
<td>0</td>
</tr>
<tr>
<td>Handler_read_rnd</td>
<td>0</td>
</tr>
<tr>
<td>Handler_read_rnd_next</td>
<td>2407004</td>
</tr>
<tr>
<td>Handler_rollback</td>
<td>0</td>
</tr>
<tr>
<td>Handler_savepoint</td>
<td>0</td>
</tr>
<tr>
<td>Handler_savepoint_rollback</td>
<td>0</td>
</tr>
<tr>
<td>Handler_update</td>
<td>0</td>
</tr>
<tr>
<td>Handler_write</td>
<td>2407001</td>
</tr>
</tbody>
</table>

15 rows in set (0.00 sec)

- **Handler_read_first**: “The number of times the first entry in an index was read.”
- **Handler_read_key**: “The number of requests to read a row based on a key.”
- **Handler_read_next**: “The number of requests to read the next row in key order.”
- **Handler_read_rnd**: “The number of requests to read the next row in the data file.”
- **Handler_read_rnd_next**: “The number of requests to insert a row in a table.”

SHOW PROFILES

- SET profiling = 1;
- .. run query ..
- SHOW PROFILES;

<table>
<thead>
<tr>
<th>Query_ID</th>
<th>Duration</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>211.21064300</td>
<td>select STRAIGHT_JOIN count(*) as c, person_id FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON (cast_info.movie_id=title.id) WHERE title.kind_id = 1 GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1</td>
</tr>
</tbody>
</table>

- show profile for query 1;
SHOW PROFILES (cont.)

```sql
mysql> show profile for query 1;
+------------------------------+------------+
| Status                       | Duration   |
+------------------------------+------------+
| starting                     |   0.002133 |
| checking permissions         |   0.000009 |
| checking permissions         |   0.000009 |
| Opening tables               |   0.000035 |
| System lock                  |   0.000022 |
| init                         |   0.000033 |
| optimizing                   |   0.000020 |
| statistics                   |   0.000032 |
| preparing                    |   0.000031 |
| Creating tmp table           |   0.000032 |
| Sorting for group            |   0.000021 |
| executing                    |   0.000005 |
+------------------------------+------------+
```

..

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Copying to tmp table</td>
<td>113.862209</td>
</tr>
<tr>
<td>converting HEAP to MyISAM</td>
<td>0.200272</td>
</tr>
<tr>
<td>Copying to tmp table on disk</td>
<td>96.506704</td>
</tr>
<tr>
<td>Sorting result</td>
<td>0.634087</td>
</tr>
<tr>
<td>Sending data</td>
<td>0.000047</td>
</tr>
<tr>
<td>end</td>
<td>0.000006</td>
</tr>
<tr>
<td>removing tmp table</td>
<td>0.004839</td>
</tr>
<tr>
<td>end</td>
<td>0.000016</td>
</tr>
<tr>
<td>query end</td>
<td>0.000004</td>
</tr>
<tr>
<td>freeing items</td>
<td>0.000064</td>
</tr>
<tr>
<td>logging slow query</td>
<td>0.000004</td>
</tr>
<tr>
<td>logging slow query</td>
<td>0.000003</td>
</tr>
<tr>
<td>cleaning up</td>
<td>0.000006</td>
</tr>
</tbody>
</table>
```

25 rows in set (0.00 sec)
Slow Log Statistics

- SET GLOBAL long_query_time = 0;
- SET GLOBAL log_slowverbosity = 'full';

```
# Time: 100924 13:58:47
# User@Host: root[root] @ localhost []
# Thread_id: 10  Schema: imdb  Last_errno: 0  Killed: 0
# Query_time: 399.563977  Lock_time: 0.000110  Rows_sent: 1  Rows_examined: 46313608
Rows_affected: 0  Rows_read: 1
# Bytes_sent: 131  Tmp_tables: 1  Tmp_disk_tables: 1  Tmp_table_sizes: 25194923
# InnoDBtrx_id: 1403
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: Yes  Tmp_table_on_disk: Yes
# Filesort: Yes  Filesort_on_disk: Yes  Merge_passes: 5
# InnoDBIO_r_ops: 1064749  InnoDBIO_r_bytes: 17444847616  InnoDBIO_r_wait: 26.935662
# InnoDB_rec_lock_wait: 0.000000  InnoDB_queue_wait: 0.000000
# InnoDB_pages_distinct: 65329

SET timestamp=1285336727;
select STRAIGHT_JOIN count(*) as c, person_id FROM cast_info FORCE INDEX(person_id)
INNER JOIN title ON (cast_info.movie_id=title.id) WHERE title.kind_id = 1 GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1;
```

This was executed on a machine with entirely cold caches.
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
Join Analysis

```sql
1. mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
2. ON (name.id = person_info.person_id) AND name.name = 'Bana, Eric'
3. 1. row
   id: 1
   select_type: SIMPLE
   table: person_info
   type: ALL
   possible_keys: NULL
   key: NULL
   key_len: NULL
   ref: NULL
   rows: 1885354
   Extra:
4. 2. row
   id: 1
   select_type: SIMPLE
   table: name
   type: eq_ref
   possible_keys: PRIMARY
   key: PRIMARY
   key_len: 4
   ref: imdb.person_info.person_id
   rows: 1
   Extra: Using where
```

Filter out as much as possible first, you can only do this by looking at WHERE clause.
Join Analysis

```
1. mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
2. ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
3. ************************* 1. row *************************
4.       id: 1
5.   select_type: SIMPLE
6.    table: name
7.     type: ref
8.   possible_keys: PRIMARY,name
9.    key: name
10.  key_len: 152
11.    ref: const
12.   rows: 1
13.  Extra: Using where
14. ************************* 2. row *************************
15.       id: 1
16.   select_type: SIMPLE
17.    table: person_info
18.     type: ALL
19.   possible_keys: NULL
20.    key: NULL
21.  key_len: NULL
22.    ref: NULL
23.   rows: 1885354
24.  Extra: Using where; Using join buffer
25. 2 rows in set (0.00 sec)

ALTER TABLE name ADD INDEX (name(50));
```
Join Analysis

```
1. mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
2. ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
3. ************************** 1. row **************************
   4. id: 1
   5. select_type: SIMPLE
   6. table: name
   7. type: ref
   8. possible_keys: PRIMARY,name
   9. key: name
10. key_len: 152
11. ref: const
12. rows: 1
13. Extra: Using where
14. ************************** 2. row **************************
   15. id: 1
   16. select_type: SIMPLE
   17. table: person_info
   18. type: ref
19. possible_keys: person_id
20. key: person_id
21. key_len: 4
22. ref: imdb.name.id
23. rows: 2
24. Extra: 
25. 2 rows in set (0.01 sec)
```

ALTER TABLE
person_info ADD INDEX (person_id);
The order you see these tables mentioned is the order MySQL has decided to join on.
Filter out as much as possible first, you can only do this by looking at WHERE clause.

```sql
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info ON name.id=cast_info.person_id INNER JOIN char_name ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
```

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>type</th>
<th>possible_keys</th>
<th>key_len</th>
<th>key</th>
<th>ref</th>
<th>rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>cast_info</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>22743540</td>
</tr>
<tr>
<td>2</td>
<td>SIMPLE</td>
<td>name</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>4</td>
<td>PRIMARY</td>
<td>imdb.cast_info.person_id</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>SIMPLE</td>
<td>char_name</td>
<td>eq_ref</td>
<td>PRIMARY</td>
<td>4</td>
<td>PRIMARY</td>
<td>imdb.cast_info.person_role_id</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
First Index:

```
mysql> ALTER TABLE char_name ADD index name_idx (name(50));
```
The order changed. `cast_info` was previously first!

Filter out as much as possible first, you can only do this by looking at `WHERE` clause.
Second Index:

```sql
mysql> ALTER TABLE cast_info ADD INDEX person_role_id_person_id(person_role_id, person_id);
```
TIP: Using a covering index means that we retrieve all data directly from the index.
Join Methods

- You need to filter as fast as possible. Here's why:
  - MySQL only uses one join method - a nested loop join.
Sample Query

• Find all actors that were active between 1960 and 1970:

**Actors:**

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
</tr>
</tbody>
</table>

**Movies:**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dr. No</td>
<td>1962</td>
</tr>
<tr>
<td>2</td>
<td>From Russia with Love</td>
<td>1963</td>
</tr>
<tr>
<td>3</td>
<td>Goldfinger</td>
<td>1964</td>
</tr>
<tr>
<td>3</td>
<td>You only live twice</td>
<td>1967</td>
</tr>
<tr>
<td>5</td>
<td>On Her Majesty's Secret Service</td>
<td>1969</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Sample Query

- Find all actors that were active between 1960 and 1970:

**Actors:**

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
</tr>
</tbody>
</table>

**Movies:**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dr. No</td>
<td>1962</td>
</tr>
<tr>
<td>2</td>
<td>From Russia with Love</td>
<td>1963</td>
</tr>
<tr>
<td>3</td>
<td>Goldfinger</td>
<td>1964</td>
</tr>
<tr>
<td>3</td>
<td>You only live twice</td>
<td>1967</td>
</tr>
<tr>
<td>5</td>
<td>On Her Majesty's Secret Service</td>
<td>1969</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
### Sample Query

- **Find all actors that were active between 1960 and 1970:**

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
</tr>
</tbody>
</table>

### Actors:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dr. No</td>
<td>1962</td>
</tr>
<tr>
<td>2</td>
<td>From Russia with Love</td>
<td>1963</td>
</tr>
<tr>
<td>3</td>
<td>Goldfinger</td>
<td>1964</td>
</tr>
<tr>
<td>3</td>
<td>You only live twice</td>
<td>1967</td>
</tr>
<tr>
<td>5</td>
<td>On Her Majesty's Secret Service</td>
<td>1969</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
## Sample Query

- Find all actors that were active between 1960 and 1970:

### Actors:

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
</tr>
</tbody>
</table>

### Movies:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dr. No</td>
<td>1962</td>
</tr>
<tr>
<td>2</td>
<td>From Russia with Love</td>
<td>1963</td>
</tr>
<tr>
<td>3</td>
<td>Goldfinger</td>
<td>1964</td>
</tr>
<tr>
<td>3</td>
<td>You only live twice</td>
<td>1967</td>
</tr>
<tr>
<td>5</td>
<td>On Her Majesty's Secret Service</td>
<td>1969</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
# Sample Query

- Find all actors that were active between 1960 and 1970:

<table>
<thead>
<tr>
<th>Actors:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>first_name</td>
<td>last_name</td>
</tr>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Movies:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>year</td>
</tr>
<tr>
<td>1</td>
<td>Dr. No</td>
<td>1962</td>
</tr>
<tr>
<td>2</td>
<td>From Russia with Love</td>
<td>1963</td>
</tr>
<tr>
<td>3</td>
<td>Goldfinger</td>
<td>1964</td>
</tr>
<tr>
<td>3</td>
<td>You only live twice</td>
<td>1967</td>
</tr>
<tr>
<td>5</td>
<td>On Her Majesty's Secret Service</td>
<td>1969</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
If that query is common

• When you can’t filter enough on one table, bring some of the other filters from the other tables to the first one:

Actors:

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
<th>start_date</th>
<th>finish_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sean</td>
<td>Connery</td>
<td>1962</td>
<td>1971</td>
</tr>
<tr>
<td>2</td>
<td>George</td>
<td>Lazenby</td>
<td>1969</td>
<td>1969</td>
</tr>
<tr>
<td>3</td>
<td>Roger</td>
<td>Moore</td>
<td>1973</td>
<td>1985</td>
</tr>
<tr>
<td>4</td>
<td>Timothy</td>
<td>Dalton</td>
<td>1987</td>
<td>1989</td>
</tr>
<tr>
<td>5</td>
<td>Pierce</td>
<td>Brosnan</td>
<td>1995</td>
<td>2002</td>
</tr>
<tr>
<td>6</td>
<td>Daniel</td>
<td>Craig</td>
<td>2006</td>
<td>2011</td>
</tr>
</tbody>
</table>
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
Subquery Analysis

Will it fix it if we add an index on `title.kind_id`?
With index on title.kind_id

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
2. (SELECT id FROM kind_type WHERE kind='video game')
3. **************************************** 1. row ****************************************
4.     id: 1
5.   select_type: PRIMARY
6.     table: title
7.   type: ALL
8.  possible_keys: NULL
9.    key: NULL
10. key_len: NULL
11. ref: NULL
12. rows: 1574389
13. Extra: Using where
14. **************************************** 2. row ****************************************
15.     id: 2
16.   select_type: DEPENDENT SUBQUERY
17.     table: kind_type
18.   type: const
19.  possible_keys: PRIMARY,kind_id
20.    key: kind_id
21. key_len: 47
22. ref: const
23. rows: 1
24. Extra: Using index
25. 2 rows in set (0.11 sec)
```
No! It doesn’t. Why is this?
Scalar Subquery

mysql> EXPLAIN SELECT * FROM title WHERE kind_id =
(_SELECT id FROM kind_type WHERE kind='video game')

*************** 1. row ***************
  id: 1
  select_type: PRIMARY
  table: title
  type: ref
  possible_keys: k
  key: k
  key_len: 4
  ref: const
  rows: 8502
  Extra: Using where

*************** 2. row ***************
  id: 2
  select_type: SUBQUERY
  table: kind_type
  type: const
  possible_keys: kind_id
  key: kind_id
  key_len: 47
  ref:
  rows: 1
  Extra: Using index
2 rows in set (0.10 sec)
Solving via Join

```
1. mysql> EXPLAIN SELECT title.* FROM title INNER JOIN kind_type ON
2. (title.kind_id = kind_type.id) WHERE kind_type.kind IN ('video game')
3. ************************** 1. row **************************
4. id: 1
5. select_type: SIMPLE
6. table: kind_type
7. type: const
8. possible_keys: PRIMARY,kind_id
9. key: kind_id
10. key_len: 47
11. ref: const
12. rows: 1
13. Extra: Using index
14. ************************** 2. row **************************
15. id: 1
16. select_type: SIMPLE
17. table: title
18. type: ref
19. possible_keys: kind_id
20. key: kind_id
21. key_len: 4
22. ref: const
23. rows: 8502
24. Extra:
25. 2 rows in set (0.00 sec)
```

It’s okay to have multiple kind’s specified using this syntax.

```
ALTER TABLE title ADD KEY (kind_id);
```
Should We Completely Avoid Them?

• No, Benchmark!

• “Delayed Join”
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
The problem with this schema, is there's just a couple of outliers with really long names:

```sql
1. mysql> SELECT max(length(title)) from title;
2. +------------+
3. | max(length(title)) |
4. +----------------+ 334 |
5. +----------------+
6. +----------------+
7. 1 row in set (6.64 sec)
8. mysql> SELECT max(length(name)) from char_name;
9. +------------+
10. | max(length(name)) |
11. +----------------+ 478 |
12. +----------------+
13. +----------------+
14. 1 row in set (7.80 sec)
```
Two ways to solve this:

1. Pick a good length to get a lot of uniqueness:

```sql
1. mysql> SELECT count(distinct(title)) as n_unique,
2. count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
3. count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
4. count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
5. count(distinct(LEFT(title, 10))) as n10  FROM title;
```
Two ways to solve this:

1. Pick a good length to get a lot of uniqueness:

```sql
mysql> SELECT count(distinct(title)) as n_unique,
2. count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
3. count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
4. count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
5. count(distinct(LEFT(title, 10))) as n10  FROM title;
```

<table>
<thead>
<tr>
<th>n_unique</th>
<th>n100</th>
<th>n75</th>
<th>n50</th>
<th>n40</th>
<th>n30</th>
<th>n20</th>
<th>n10</th>
</tr>
</thead>
<tbody>
<tr>
<td>998335</td>
<td>998320</td>
<td>998291</td>
<td>997887</td>
<td>996727</td>
<td>991532</td>
<td>960894</td>
<td>624949</td>
</tr>
</tbody>
</table>
Two ways to solve this:

1. Pick a good length to get a lot of uniqueness:

   1. mysql> SELECT count(distinct(title)) as n_unique,
   2. count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
   3. count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
   4. count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
   5. count(distinct(LEFT(title, 10))) as n10 FROM title;

<table>
<thead>
<tr>
<th>n_unique</th>
<th>n100</th>
<th>n75</th>
<th>n50</th>
<th>n40</th>
<th>n30</th>
<th>n20</th>
<th>n10</th>
</tr>
</thead>
<tbody>
<tr>
<td>998335</td>
<td>998320</td>
<td>998291</td>
<td>997887</td>
<td>996727</td>
<td>991532</td>
<td>960894</td>
<td>624949</td>
</tr>
</tbody>
</table>

   96% uniqueness, but only 20 chars instead of 300+
   Looks pretty good to me:
   ALTER TABLE title ADD index (name(20))
Option 2: Emulate a Hash Index

Is possible only with MEMORY engine:

```
ALTER TABLE table ADD INDEX USING HASH (title);
```
Option 2: Emulate a Hash Index

<table>
<thead>
<tr>
<th></th>
<th>count(distinct(BINARY title))</th>
<th>count(distinct(title_crc32))</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>1001509</td>
<td>1001393</td>
</tr>
</tbody>
</table>

A good hashing algorithm has good distribution. How good is this?
Option 2: Hash Index (cont.)

- Query needs to be transformed slightly to:
  ```sql
  SELECT * FROM title WHERE
  title_crc32=crc32('my_title') AND
  title='my_title';
  ```

- All updates/inserts need to also update the value of title_crc32:
  ```sql
  SELECT * FROM title WHERE
  title_crc32=crc32('my_title') AND
  title='my_title';
  ```

  Can be easily done via the application, or a trigger if write load is very low.
Pros/Cons

Prefix Index:

★ **Pro:**
Built in to MySQL/no magic required.

★ **Cons:**
Not very effective when the start of the string is not very unique.

Hash Index:

★ **Pro:**
Very Good when there is not much uniqueness until very far into the string.

★ **Cons:**
Equality searches only. Requires ugly magic to work with collations/case sensitivity.
Things are looking good!?

- Please *don’t* take away that adding indexes == only secret to performance.
  - The story is a lot more complicated.
  - We have to get around optimizer limitations, and a lack of index/join options.
Optimizer Hints

- Optimizer decision making is all about tradeoffs.
  - MySQL wants to pick the best plan, but it can’t be exhaustive in deciding if it takes too long.

- If MySQL is off by a lot, you may want to provide a hint:
  - USE INDEX
  - FORCE INDEX
  - IGNORE INDEX
  - STRAIGHT_JOIN

Optimizer Hints, Should We?

```sql
1. mysql> explain SELECT title.* FROM title force key (kind_id_title)
2. INNER JOIN kind_type ON (title.kind_id = kind_type.id)
3. WHERE kind_type.kind IN ('video game')\G
4. ************************************************** 1. row **************************************************
   5. id: 1
   6. select_type: SIMPLE
   7. table: kind_type
   8. type: const
   9. possible_keys: PRIMARY,kind
   10. key: kind
   11. key_len: 47
   12. ref: const
   13. rows: 1
   14. Extra: Using index
5. ************************************************** 2. row **************************************************
   15. id: 1
16. select_type: SIMPLE
17. table: title
18. type: ref
19. possible_keys: kind_id_title
20. key: kind_id_title
21. key_len: 4
22. ref: const
23. rows: 14240
24. Extra:
More features & workarounds

- `EXPLAIN` only works for `SELECT`: convert `UPDATE/DELETE` to `SELECT` (feature added in 5.6)
- The IN() list workaround
- Index Merge
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
CREATE TABLE

1. mysql> SHOW CREATE TABLE title\G
2. *******************************************************************  1. row *******************************************************************
3.     Table: title
4. Create Table: CREATE TABLE `title` (  
5.     `id` int(11) NOT NULL AUTO_INCREMENT,
6.     `title` text NOT NULL,
7.     `imdb_index` varchar(12) DEFAULT NULL,
8.     `kind_id` int(11) NOT NULL,
9.     `production_year` int(11) DEFAULT NULL,
10.    `imdb_id` int(11) DEFAULT NULL,
11.    `phonetic_code` varchar(5) DEFAULT NULL,
12.    `episode_of_id` int(11) DEFAULT NULL,
13.    `season_nr` int(11) DEFAULT NULL,
14.    `episode_nr` int(11) DEFAULT NULL,
15.    `series_years` varchar(49) DEFAULT NULL,
16.    `title_crc32` int(10) unsigned DEFAULT NULL,
17.    PRIMARY KEY (`id`),
18.    KEY `kind_id` (`kind_id`),
19.    KEY `title_3` (`title`(255)),
SHOW FIELDS

```
mysql> SHOW COLUMNS FROM title;
+-----------------+-----------------------------+--------+--------+-----------------+-------------------+
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>int(11)</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
</tr>
<tr>
<td>title</td>
<td>text</td>
<td>NO</td>
<td>MUL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>imdb_index</td>
<td>varchar(12)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>kind_id</td>
<td>int(11)</td>
<td>NO</td>
<td>MUL</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>production_year</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>imdb_id</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>phonetic_code</td>
<td>varchar(5)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>episode_of_id</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>season_nr</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>episode_nr</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>series_years</td>
<td>varchar(49)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>title_crc32</td>
<td>int(10) unsigned</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
+-----------------+-----------------+--------+--------+-----------------+-------------------+
12 rows in set (0.00 sec)
```
SHOW INDEXES

<table>
<thead>
<tr>
<th>Table</th>
<th>nonuniq</th>
<th>Key_name</th>
<th>seq</th>
<th>Column_name</th>
<th>coll</th>
<th>card</th>
<th>Sub_part</th>
<th>Packed</th>
<th>Null</th>
<th>Index_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>0</td>
<td>PRIMARY</td>
<td>1</td>
<td>id</td>
<td>A</td>
<td>1542249</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>BTREE</td>
</tr>
<tr>
<td>title</td>
<td>1</td>
<td>kind_id</td>
<td>1</td>
<td>kind_id</td>
<td>A</td>
<td>204</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>BTREE</td>
</tr>
<tr>
<td>title</td>
<td>1</td>
<td>title_3</td>
<td>1</td>
<td>title</td>
<td>A</td>
<td>204</td>
<td>255</td>
<td>NULL</td>
<td></td>
<td>BTREE</td>
</tr>
<tr>
<td>title</td>
<td>1</td>
<td>kind_id_title</td>
<td>1</td>
<td>kind_id</td>
<td>A</td>
<td>204</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>BTREE</td>
</tr>
<tr>
<td>title</td>
<td>1</td>
<td>kind_id_title</td>
<td>2</td>
<td>title</td>
<td>A</td>
<td>204</td>
<td>50</td>
<td>NULL</td>
<td></td>
<td>BTREE</td>
</tr>
</tbody>
</table>

5 rows in set (0.00 sec)
Day One Advice (1)

- Keep it simple - store atomic types in each field.
  - This means storing first name and last name as two separate fields.

- Don’t try and get tricky with how you store the data.
  - i.e. this field means it’s a phone number unless this other field is set to something.
Day One Advice (2)

- Use appropriate data types -
  - If you’re not sure about the length, varchar(100) is still much better than varchar(255).
  - Use an Integer for a number. Decimal for precision numbers, float for non-precision numbers, etc.
  - If integers don’t have to be negative, use unsigned.
Day One Advice (3)

- Plan how you will be accessing the data.
  - If you know that you have to do 4 expensive joins to execute a common query - it might not be the best solution.
  - It’s okay to have redundancy in the database from very early on. One example is pre-generating the ‘average score’ on IMDB titles.
Clustered Index

• Everything in InnoDB is an index:
  • Data is stored in a clustered index organized by the primary key. In the absence of a primary key, the first unique not null key is selected*.
  • Other indexes are stored in secondary indexes.

* In the absence of a unique key, a hidden 6 byte key is created.
What is a clustered index?

- First let's look at how MyISAM stores data*:

<table>
<thead>
<tr>
<th>Staff.MYI</th>
<th>Staff.MYD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
<td><strong>First Name</strong></td>
</tr>
<tr>
<td>8</td>
<td>lePeter</td>
</tr>
<tr>
<td>4 12</td>
<td>leVadim</td>
</tr>
<tr>
<td>2 6 10 14</td>
<td>leFred</td>
</tr>
<tr>
<td>1 3 5 7 9 11 13 15</td>
<td>leEwen</td>
</tr>
<tr>
<td></td>
<td>leBaron</td>
</tr>
<tr>
<td></td>
<td>..</td>
</tr>
</tbody>
</table>

Data is stored “roughly” in insertion order, with no guarantees, i.e.

Deleted rows may be filled with newer records.

* Illustrating B-Tree as Binary Tree for simplicity
What is a clustered index (cont.)

- A MyISAM primary key lookup looks something like this:

<table>
<thead>
<tr>
<th>Staff.MYI</th>
<th>Staff.MYD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
What is a clustered index (cont.)

• A MyISAM primary key lookup looks something like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lePeter</td>
</tr>
<tr>
<td>2</td>
<td>leVadim</td>
</tr>
<tr>
<td>7</td>
<td>leFred</td>
</tr>
<tr>
<td>4</td>
<td>leEwen</td>
</tr>
<tr>
<td>5</td>
<td>leBaron</td>
</tr>
</tbody>
</table>

Traverse the index to find the address of the row we are looking for.
A MyISAM primary key lookup looks something like this:

- Traverse the index to find the address of the row we are looking for.
- Lookup the address in the data file.

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lePeter</td>
</tr>
<tr>
<td>2</td>
<td>leVadim</td>
</tr>
<tr>
<td>3</td>
<td>leFred</td>
</tr>
<tr>
<td>4</td>
<td>leEwen</td>
</tr>
<tr>
<td>5</td>
<td>leBaron</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
What is a clustered index (cont.)

* An InnoDB Primary Key lookup looks like this:

Staff.ibd

* Illustrating B+Tree as Binary Tree for simplicity.
An InnoDB Primary Key lookup looks like this:

* Illustrating B+Tree as Binary Tree for simplicity.
What is a clustered index (cont.)

* An InnoDB Primary Key lookup looks like this:

```
+-----+---------------+---------------+---------------+---------------+---------------+---------------+
|  1  |   3           |     5         |     7         |     9         |     11        |     13        |
|  12 | 0xACDC        | 12 0xACDC     | 12 0xACDC     | 12 0xACDC     | 12 0xACDC     | 12 0xACDC     |
+-----+---------------+---------------+---------------+---------------+---------------+---------------+
```

* Illustrating B+Tree as Binary Tree for simplicity.

Staff.ibd

* Traverse the index to find the full row.

leVadim, ..., ...
An InnoDB Primary Key lookup looks like this:

- Illustrating B+Tree as Binary Tree for simplicity.

Traverse the index to find the full row.

Stop here.

* Illustrating B+Tree as Binary Tree for simplicity.
What is a clustered index (cont.)

* A secondary key lookup looks like this:

```
extension_number

8
4  12
2  6  10  14
1  3  5  7  9  11  13  15
```

```
12 0xACDC
4 0xACDC 12 0xACDC
2 0xACDC 6 0xACDC 10 0xACDC 14 0xACDC
1 ... 3 ... 5 ... 7 ... 9 ... 11 ... 13 ... 15 ...
```
A secondary key lookup looks like this:

<table>
<thead>
<tr>
<th>extension_number</th>
<th>4</th>
<th>6</th>
<th>10</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Traverse the index to find the value of the primary key.

<table>
<thead>
<tr>
<th>12</th>
<th>0xACDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0xACDC</td>
</tr>
<tr>
<td>10</td>
<td>0xACDC</td>
</tr>
<tr>
<td>14</td>
<td>0xACDC</td>
</tr>
</tbody>
</table>

| 1  | ..   | 3  | .. | 5  | .. | 7  | .. | 9  | .. | 11 | .. | 13 | .. | 15 | .. |
A secondary key lookup looks like this:

Traverse the index to find the value of the primary key.

Traverse the primary key to find the full row.
Clustered Index (cont.)

This design has some interesting consequences:

- Primary key lookups are very fast.
- Inserting data in order is fast - out of order can be very slow, and cause fragmentation.
- Secondary indexes can become very large if you have a large primary key.
In practical terms this means:

- Don’t use GUIDs for InnoDB tables!
- Never piggy-back the primary key index into the end of a composite index or covering index - it is already included for free.
Some MySQL(InnoDB) Specifics

- **Primary Keys:**
  - Always specify one.
  - Keep it short.
  - Try and make it your primary access method.
  - Keep insertion incremental.

- **Composite Keys:**
  - Don’t ever include the primary key index as part of a covering index.
Our results (typical case)

CREATE TABLE uuid_users (  
    PRIMARY KEY,  
    emailaddress varchar(100),  
    firstname varchar(20),  
    lastname varchar(20),  
    birthday varchar(10),  
    occupation varchar(70),  
    INDEX(emailaddress),  
    INDEX(lastname, firstname),  
    INDEX(occupation)  
) ENGINE=InnoDB;

The UUID primary key makes the table about 65% larger.
CREATE TABLE mydata (  
PRIMARY KEY,  
col1 INT NOT NULL,  
col2 INT NOT NULL,  
col3 INT NOT NULL,  
col4 INT NOT NULL,  
col5 INT NOT NULL,  
INDEX (col1),  
INDEX (col2),  
INDEX (col3),  
INDEX (col4),  
INDEX (col5)  
) ENGINE=InnoDB;

The UUID primary key makes the table almost x3!
CREATE TABLE users (  
  ID INTEGER,  
  first_name VARCHAR(60),  
  last_name VARCHAR(60),  
  email VARCHAR(100),  
  phone_number VARCHAR(20),  
  last_login_date DATE  
);
Hot column on a wide table

- Solutions & Workarounds:
  - Move `user_id` and `last_login_date` to another table (good for reads and writes).
  - Use a covering index (better for situations where read heavy).
Hot column on a wide table

• Another example of this problem is with the ‘view count’ on an item.
• For this, writing to memcached and only pushing down to MySQL on every \textit{nth} write may be required.
  • Denormalization might not buy you enough time.
Over-indexed tables

- Infrequently used indexes can be responsible for decreasing write capacity.
  - more data in buffer pool
  - more disk IO
  - more time to update
- For reads, the optimizer has more choices to make and a more difficult decision process.
Under-indexed Tables

• Under-indexed tables can result in too many rows needing to be examined after an index has been used - or in the worst case, no index used.

  • This can cause contention on what contents you are able to keep in memory - and it will likely increase the size of your working set.
What makes a good schema?
What is good?

- It all depends on the queries you send to it.
  - i.e. if you can’t add a very effective index, you need to make changes.
Best way to Design Schema

• Use a program where you can map out each of the objects on an ER diagram.
  • i.e. MySQL Workbench.

• Think ahead and see if there are any common access patterns which do not fit well.
  • i.e. I always want to know the total amount of the invoice without having to sum up all the invoice items.

• Export the ER diagram to SQL.
Can you make schema better?

- It is very hard to retrofit into an Application.
- For some obvious bad-choices, the ‘band aid’ approach may work.
  - This command shows the most optimal data type:
    ```sql
    SELECT * FROM title PROCEDURE ANALYSE(1,1)\G
    ```
### PROCEDURE ANALYZE

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Min Value</th>
<th>Max Value</th>
<th>Min Length</th>
<th>Max Length</th>
<th>Empties_or_zeros</th>
<th>Nulls</th>
<th>Avg Value_or_avg_length</th>
<th>Std</th>
<th>Optimal_fieldtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>imdb.title.id</td>
<td>1</td>
<td>1543720</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>771860.5411</td>
<td>891266.7873</td>
<td>MEDIUMINT(7) UNSIGNED NOT NULL</td>
</tr>
<tr>
<td>imdb.title.title</td>
<td># 1997 Honda Accord: Gauges Upgrade - Dröng sýn</td>
<td># 1997 Honda Accord: Gauges Upgrade - Dröng sýn</td>
<td>1</td>
<td>334</td>
<td>0</td>
<td>0</td>
<td>16.4844</td>
<td>NULL</td>
<td>TEXT NOT NULL</td>
</tr>
</tbody>
</table>
MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
Identifying Bad Queries

- Slowlog
- Logging in the Application
- *-Proxy
- MySQL Query Analyzer


pt-query-digest

pt-query-digest

- generate reports from
  - slow query log
  - binlog files
  - processlist
  - postgresql log files
  - general log (not so useful)
  - tcpdump files that captured traffic from: mysql, memcached, http
- store reports in db: --review, --review-history
- enhanced filtering capabilities
  \$event->\{fingerprint\} =~ m/^select/
# 834.7s user time, 9.1s system time, 302.78M rss, 392.96M vsz
# Current date: Mon Nov 29 09:47:43 2010
# Hostname: servername
# Files: STDIN
# Overall: 670.66k total, 1.73k unique, 955.33 QPS, 3.08x concurrency
# Time range: 2010-11-29 09:14:29.955239 to 09:26:11.979320

<table>
<thead>
<tr>
<th>Attribute</th>
<th>total</th>
<th>min</th>
<th>max</th>
<th>avg</th>
<th>95%</th>
<th>stddev</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec time</td>
<td>2163s</td>
<td>0</td>
<td>3s</td>
<td>3ms</td>
<td>2ms</td>
<td>29ms</td>
<td>89us</td>
</tr>
<tr>
<td>Rows affecte</td>
<td>18.58k</td>
<td>0</td>
<td>146</td>
<td>0.03</td>
<td>0</td>
<td>0.49</td>
<td>0</td>
</tr>
<tr>
<td>Query size</td>
<td>121.29M</td>
<td>6</td>
<td>21.55k</td>
<td>189.64</td>
<td>363.48</td>
<td>328.74</td>
<td>97.36</td>
</tr>
<tr>
<td>Warning coun</td>
<td>438.18k</td>
<td>0</td>
<td>25.60k</td>
<td>0.67</td>
<td>0</td>
<td>122.19</td>
<td>0</td>
</tr>
</tbody>
</table>

# Boolean:

- No good inde: 0% yes, 99% no
- No index use: 10% yes, 89% no
<table>
<thead>
<tr>
<th>Rank</th>
<th>Query ID</th>
<th>Response time</th>
<th>Calls</th>
<th>R/Call</th>
<th>Apdx</th>
<th>V/M</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x3928FBFF36663F33</td>
<td>1349.6240</td>
<td>11976</td>
<td>0.1127</td>
<td>1.00</td>
<td>0.03</td>
<td>SELECT loan_officer_states</td>
</tr>
<tr>
<td>2</td>
<td>0x8A539A15CDC891EB</td>
<td>114.9014</td>
<td>437</td>
<td>0.2629</td>
<td>1.00</td>
<td>0.50</td>
<td>SELECT processing_assigned</td>
</tr>
<tr>
<td>3</td>
<td>0xFA5D75AB1925777C</td>
<td>92.9441</td>
<td>791</td>
<td>0.1175</td>
<td>1.00</td>
<td>0.06</td>
<td>SELECT security_dashboard</td>
</tr>
<tr>
<td>4</td>
<td>0x6F1DB5CAB019DB16</td>
<td>77.5712</td>
<td>43</td>
<td>1.8040</td>
<td>0.65</td>
<td>0.73</td>
<td>SELECT history assigned</td>
</tr>
<tr>
<td>5</td>
<td>0xDFEC78D47187A0CD</td>
<td>67.1673</td>
<td>296</td>
<td>0.2269</td>
<td>1.00</td>
<td>0.17</td>
<td>SELECT history assigned</td>
</tr>
<tr>
<td>6</td>
<td>0x5D51E5F01B88B79E</td>
<td>49.0330</td>
<td>15630</td>
<td>0.0031</td>
<td>1.00</td>
<td>0.00</td>
<td>ADMIN CONNECT</td>
</tr>
<tr>
<td>7</td>
<td>0xD704F6F4D36804AB</td>
<td>43.4990</td>
<td>274</td>
<td>0.1588</td>
<td>1.00</td>
<td>0.12</td>
<td>SELECT user_agents</td>
</tr>
<tr>
<td>8</td>
<td>0x7EC8CF8EAFCC6907</td>
<td>30.0898</td>
<td>416</td>
<td>0.0723</td>
<td>1.00</td>
<td>0.07</td>
<td>SELECT security_dashboard</td>
</tr>
<tr>
<td>9</td>
<td>0x599BEF84DBA12853</td>
<td>19.6506</td>
<td>13424</td>
<td>0.0015</td>
<td>1.00</td>
<td>0.01</td>
<td>UPDATE user_sessions</td>
</tr>
<tr>
<td>10</td>
<td>0x19EE1A48A2B249</td>
<td>18.8828</td>
<td>54835</td>
<td>0.0003</td>
<td>1.00</td>
<td>0.00</td>
<td>SELECT leads contact_info</td>
</tr>
<tr>
<td>11</td>
<td>0xDD930BC5FC65A135</td>
<td>18.6386</td>
<td>54975</td>
<td>0.0003</td>
<td>1.00</td>
<td>0.00</td>
<td>SELECT history</td>
</tr>
<tr>
<td>12</td>
<td>0x277A0E5B9646746B</td>
<td>16.2016</td>
<td>55280</td>
<td>0.0003</td>
<td>1.00</td>
<td>0.00</td>
<td>SELECT history</td>
</tr>
<tr>
<td>13</td>
<td>0x522C69BD415338C6</td>
<td>13.5388</td>
<td>300</td>
<td>0.0451</td>
<td>1.00</td>
<td>0.02</td>
<td>SELECT history assigned</td>
</tr>
<tr>
<td>14</td>
<td>0xA018F3BA9E66B42B</td>
<td>13.5138</td>
<td>41</td>
<td>0.3296</td>
<td>1.00</td>
<td>0.00</td>
<td>SELECT new_rate_locks</td>
</tr>
<tr>
<td>15</td>
<td>0x59F9E8645FFF4A16</td>
<td>12.7311</td>
<td>55331</td>
<td>0.0002</td>
<td>1.00</td>
<td>0.00</td>
<td>SELECT realtor_leads</td>
</tr>
<tr>
<td>16</td>
<td>0xEE18B363E8DB0222</td>
<td>10.6596</td>
<td>161</td>
<td>0.0662</td>
<td>1.00</td>
<td>0.11</td>
<td>SELECT nb_alert_notes</td>
</tr>
<tr>
<td>17</td>
<td>0xDF78E27C3290E5F2</td>
<td>10.2883</td>
<td>345</td>
<td>0.0298</td>
<td>1.00</td>
<td>0.01</td>
<td>SELECT history lo_history</td>
</tr>
<tr>
<td>18</td>
<td>0xC82802FC73439D3</td>
<td>10.0459</td>
<td>9</td>
<td>1.1162</td>
<td>0.67</td>
<td>0.20</td>
<td>SELECT users help_history</td>
</tr>
<tr>
<td>19</td>
<td>0x5462226BD2AF82D9</td>
<td>7.1391</td>
<td>75</td>
<td>0.0952</td>
<td>1.00</td>
<td>0.16</td>
<td>SELECT tasks task_note</td>
</tr>
<tr>
<td>20</td>
<td>0x177159F6BEA4126A</td>
<td>6.7048</td>
<td>55342</td>
<td>0.0001</td>
<td>1.00</td>
<td>0.01</td>
<td>SELECT nb_alert_notes</td>
</tr>
<tr>
<td>MISC</td>
<td>0xMISC</td>
<td>179.8054</td>
<td>350684</td>
<td>0.0005</td>
<td>NS</td>
<td>0.0</td>
<td>&lt;1713 ITEMS&gt;</td>
</tr>
</tbody>
</table>
# Query 1: 17.06 QPS, 1.92x concurrency, ID 0x3928FBFF36663F33 at byte 1417466467
# This item is included in the report because it matches --limit.
# Scores: Apdex = 1.00 [1.0], V/M = 0.03
# Time range: 2010-11-29 09:14:30.052415 to 09:26:11.914796

<table>
<thead>
<tr>
<th>Attribute</th>
<th>pct</th>
<th>total</th>
<th>min</th>
<th>max</th>
<th>avg</th>
<th>95%</th>
<th>stddev</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>1</td>
<td>11976</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exec time</td>
<td>62</td>
<td>1350s</td>
<td>25ms</td>
<td>395ms</td>
<td>113ms</td>
<td>219ms</td>
<td>54ms</td>
<td>91ms</td>
</tr>
<tr>
<td>Rows affected</td>
<td>0</td>
<td>39</td>
<td>0</td>
<td>35</td>
<td>0.00</td>
<td>0</td>
<td>0.32</td>
<td>0</td>
</tr>
<tr>
<td>Query size</td>
<td>23</td>
<td>28.75M</td>
<td>2.46k</td>
<td>2.46k</td>
<td>2.46k</td>
<td>2.38k</td>
<td>0</td>
<td>2.38k</td>
</tr>
<tr>
<td>Warning count</td>
<td>11</td>
<td>51.51k</td>
<td>0</td>
<td>12.80k</td>
<td>4.40</td>
<td>0</td>
<td>233.99</td>
<td>0</td>
</tr>
</tbody>
</table>

# Boolean:
- No index use: 99% yes, 0% no

# String:

# Databases
# Errors: none (273/99%), #1064 (1/0%)
# Hosts: 172.20.101.178
# Users: dbuser
# Query_time distribution
#   1us
#  10us ####################################
# 100us ###########
#   1ms ##
#  10ms #
# 100ms ####################################################
#    1s
#  10s+
# Tables
#    SHOW TABLE STATUS LIKE 'user_agents'\G
#    SHOW CREATE TABLE `user_agents`\G
#  EXPLAIN /*!50100 PARTITIONS*/
SELECT user_agent_id, search_engine
FROM user_agents
WHERE user_agent='Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.0.3705)'
\G
# Item 1: 3.41 QPS, 0.97x concurrency, ID 0xABCE5AD2A2DD1BA1 at byte 2881246
# This item is included in the report because it matches --limit.
# Scores: Apdex = 0.97 [1.0], V/M = 19.02
# Query_time sparkline: | ^______|
# Time range: 2011-04-05 16:12:13 to 16:14:45
# Attribute     pct  total     min     max     avg    95%    stddev  median
# ============ === ======= ======= ======= ======= ======= ======= ========
# Count          0     519
# Exec time      2    148s    11us     33s   285ms    53ms      2s    26us
# Lock time      0     5ms       0   334us     9us    66us    32us       0
# Rows sent      0      41       0       1    0.08    0.99    0.27       0
# Rows examine   1  4.97M       0 445.49k   9.80k   5.73k  49.33k       0
# Rows affecte   0     2       0       1    0.00       0    0.06       0
# Rows read      1  2.01M       0 250.47k   3.96k    1.96  27.94k    0.99
# Bytes sent     0 241.20k      11   8.01k  475.89  918.49  689.98  258.32
# Merge passes   0       0       0       0       0       0       0       0
# Tmp tables     0      15       0       1    0.03       0    0.17       0
# Tmp disk tbl   0       3       0       1    0.01       0    0.08       0
# Tmp tbl size   0   4.78k       0   4.78k    9.43       0  211.60       0
# Query size     0 100.95k      19   2.71k  199.17  363.48  206.60  151.03
# InnoDB:
# IO r bytes     0       0       0       0       0       0       0       0
# IO r ops       0       0       0       0       0       0       0       0
# IO r wait      0       0       0       0       0       0       0       0
# pages distin   1  67.99k       0  10.64k   1.26k   3.88k   2.47k   31.70
# queue wait     0       0       0       0       0       0       0       0
# rec lock wai   0       0       0       0       0       0       0       0
# Boolean:
# Filesort      0% yes, 99% no
# Full scan     7% yes, 92% no
# QC Hit        78% yes, 21% no
# Tmp table     2% yes, 97% no
# Tmp table on  0% yes, 99% no
<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tmp tbl size</td>
<td>0</td>
<td>4.78k</td>
<td>0</td>
<td>4.78k</td>
<td>9.43</td>
<td>0</td>
</tr>
<tr>
<td>Query size</td>
<td>0</td>
<td>100.95k</td>
<td>19</td>
<td>2.71k</td>
<td>199.17</td>
<td>363.48</td>
</tr>
<tr>
<td>InnoDB:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO r bytes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IO r ops</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IO r wait</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pages distin</td>
<td>1</td>
<td>67.99k</td>
<td>0</td>
<td>10.64k</td>
<td>1.26k</td>
<td>3.88k</td>
</tr>
<tr>
<td>queue wait</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rec lock wait</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Boolean:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filesort</td>
<td>0% yes,</td>
<td>99% no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full scan</td>
<td>7% yes,</td>
<td>92% no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QC Hit</td>
<td>78% yes,</td>
<td>21% no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmp table</td>
<td>2% yes,</td>
<td>97% no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmp table on</td>
<td>0% yes,</td>
<td>99% no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Is It Worth it?

• There’s always room for improvement
• Ask yourself: Is the change going to have a benefit?
• How much effort does it take to get how much gain?
• Benchmark!
• Instrument!

MySQL Query Optimization

- Basic Query Tuning
- Beyond EXPLAIN
- Optimizing JOINs
- Subquery
- Other Optimizations
- Table Schema
- Identifying Bad Queries
We're Hiring!

www.percona.com/about-us/careers/