INEA00104L: Report #2

Due on Monday, June 2, 2014

Mgr inż. Andrzej Wytyczak-Partyka

inż. Piotr Giedziun

Introduction

This report was made to present results of work done during last class. Main goal of this paper it to document different aspects of indexing performance.

Task 1

Why is MySQL choosing a full table scan instead of using index on price column when doing a SELECT * FROM tickets ORDER BY price?

Quick reminder of 'ticket' table structure.

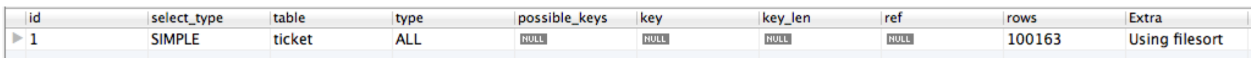
```
mysql> SHOW COLUMNS FROM sakila.ticket;
+----------------------+----------------------+------+-----+---------+----------------+
| Field                | Type                 | Null | Key | Default | Extra          |
+----------------------+----------------------+------+-----+---------+----------------+
| ticket_id            | int(11)              | NO   | PRI | NULL    | auto_increment |
| price                | float                | NO   | MUL | NULL    |                |
| seat_seat_id         | int(11)              | NO   | MUL | NULL    |                |
| show_show_id         | int(11)              | NO   | MUL | NULL    |                |
| cinema_cinema_id     | int(11)              | NO   | MUL | NULL    |                |
| customer_customer_id | smallint(5) unsigned | YES  | MUL | NULL    |                |
| theater_id           | int(11)              | NO   | MUL | NULL    |                |
+----------------------+----------------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

We were ask to add index on price column.

```
mysql> ALTER TABLE sakila.ticket ADD INDEX price_idx (price);
```

Following query should take advantage of price_idx index, with should result in full index scan.

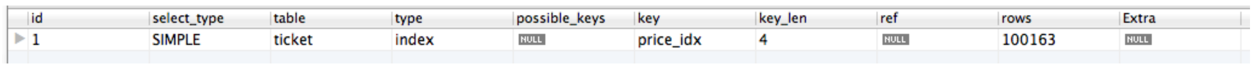
```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM sakila.ticket ORDER BY price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	ALL	NULL	NULL	NULL	NULL	100163	Using filesort

This result looks surprising, MySQL optimizer decided to do a full table scan (Using filesort, key=NULL). Let's try do another one, this time I will force optimizer to use index.

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM sakila.ticket FORCE INDEX(price_idx)
     ORDER BY price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	index	NULL	price_idx	4	NULL	100163	NULL

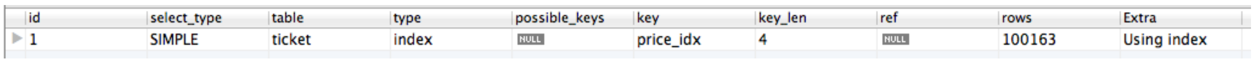
This time around we are doing full index scan (key=price_idx). Number of rows is exactly the same as previous. Both cases are performing full scan.

In order to understand the difference we have to understand, how MySQL is storing data. When we are doing full table scan, MySQL is performing sequential reads of stored data. However, when we are doing full index scan, MySQL have to perform a lot of random reads (reading data pointed by index).

Task 1 continued on next page. . .

In this case full table scan is just faster than full index scan. I did few tests on my local machine in order to confirm it. Unlucky, I have SSD drive, so random read is not a problem for this hardware architecture. MySQL optimizer decided to use full table scan by default because this method should be faster (generaly speaking).
It also explain why MySQL optimizer used full index scan for following query.

```
mysql> EXPLAIN SELECT price FROM sakila.ticket ORDER BY price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	index	NULL	price_idx	4	NULL	100163	Using index

This time around there is no need for random data access. All data we asked is stored in indexing structure. We don't have to use indexing pointer anymore.

Task 2

Performance issues involved with using indexes on VARCHAR fields. How does prefixing them influence index size and QPS

This issue was explained in my previous report.

Task 3

Explain why EXPLAIN SELECT SQL_NO_CACHE * FROM sakila.ticket ORDER BY theater_id DESC is not using theater_id_price_idx

This issue was explained in Task 1 section.

TL;DR: MySQL optimizer decided to use full table scan, because it's faster than full index scan. Sequential reads are performed faster than random reads (on default HD drive).

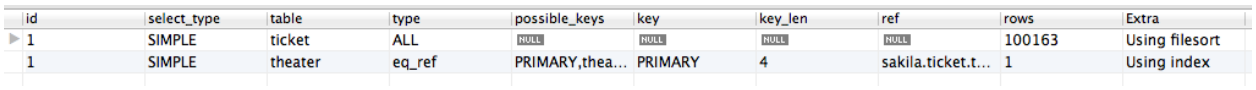
Task 4

Explain why EXPLAIN SELECT ticket.price FROM ticket LEFT JOIN theater on theater.theater_id = ticket.theater_id ORDER BY ticket.price DESC is not using price_idx INDEX.

This query is also related with Task 1. I was looking for an answer on MySQL doc, but I didn't found one. What I'm about to say is pure guess. I will try to support it with evidence.

First, let's execute following query.

```
mysql> EXPLAIN SELECT ticket.price FROM ticket LEFT JOIN theater
     on theater.theater_id = ticket.theater_id ORDER BY ticket.price;
```



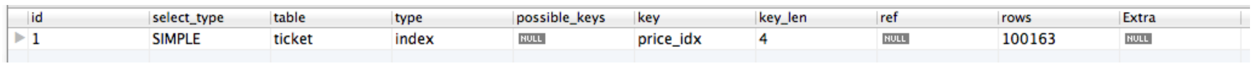
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	ALL	NULL	NULL	NULL	NULL	100163	Using filesort
1	SIMPLE	theater	eq_ref	PRIMARY,thea…	PRIMARY	4	sakila.ticket.t…	1	Using index

My first guess was, that this query will use price_idx index. Query looks similar to SELECT price FROM sakila.ticket ORDER BY price (explained in Task 1 section). MySQL decided to use full file sort, because it's

Task 4 continued on next page. . .

faster that full index sort (also explained in Task 1 section). The question is: Why it differs from SELECT price FROM sakila.ticket ORDER BY price, while using JOIN?

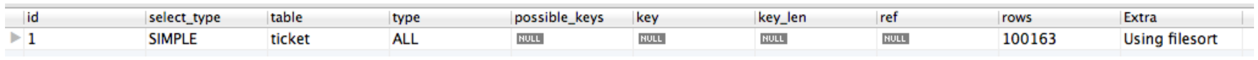
```
mysql> SELECT price FROM sakila.ticket ORDER BY price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	index	NULL	price_idx	4	NULL	100163	NULL

This query will return all prices values. This data is stored in index structure. MySQL will use full index scan, because it does not requires additional random reads (all data is already provided). However, if we will return additional field, who's not a part of index, MySQL will perform full table scan.

```
mysql> SELECT theater_id, price FROM sakila.ticket ORDER BY price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	ALL	NULL	NULL	NULL	NULL	100163	Using filesort

```
mysql> EXPLAIN SELECT ticket.price FROM ticket LEFT JOIN theater
     on theater.theater_id = ticket.theater_id ORDER BY ticket.price;
```

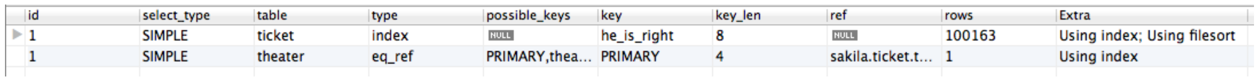
Using LEFT JOIN forced MySQL to look for ticket.theater_id. Instead of "SELECT ticket.price" it's "SELECT ticket.theater_id, ticket.price".

In order to make it more reliable, I decided to crate new index, containing both fields (price and theater_id).

```
mysql>  ALTER TABLE sakila.ticket ADD INDEX he_is_right (theater_id, price);
```

Now, let's execute JOIN query once more.

```
mysql> EXPLAIN SELECT ticket.price FROM ticket LEFT JOIN theater
     on theater.theater_id = ticket.theater_id ORDER BY ticket.price;
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ticket	index	NULL	he_is_right	8	NULL	100163	Using index; Using filesort
1	SIMPLE	theater	eq_ref	PRIMARY,thea...	PRIMARY	4	sakila.ticket.t...	1	Using index

This time MySQL optimizer used he_is_right index. It's because all data MySQL needs (theater_id and price) to perform JOIN was stored in index structure.