

# MySQL Explain Example – Explaining MySQL EXPLAIN using StackOverflow data

October 1, 2018 ·

Tomer Shay @ EverSQL

I personally believe that the best way to deliver a complicated message to an audience, is by using a simple example. So in this post, I chose to demonstrate how to obtain insights from MySQL's EXPLAIN output, by using a simple SQL query which fetches data from [StackOverflow's publicly available dataset](#).

The EXPLAIN command provides information about how MySQL executes queries. EXPLAIN can work with SELECT, DELETE, INSERT, REPLACE, and UPDATE statements.

We'll first analyze the original query, then attempt to optimize the query and look into the optimized query's execution plan to see what changed and why.

This is the first article in a series of posts. Each post will walk you through a more advanced SQL query than the previous post, while demonstrating more insights which can be obtained from MySQL's execution plans.

## The query and database structure

The following is the structure of the two tables used by this example query (*posts* and *votes*):

```

1 CREATE TABLE `posts` (
2   `Id` int(11) NOT NULL,
3   `AcceptedAnswerId` int(11) DEFAULT NULL,
4   `AnswerCount` int(11) DEFAULT NULL,
5   `Body` longtext CHARACTER SET utf8 NOT NULL,
6   ...
7   `OwnerUserId` int(11) DEFAULT NULL,
8   ...
9   `Title` varchar(250) CHARACTER SET utf8 DEFAULT NULL,
10  `ViewCount` int(11) NOT NULL
11  PRIMARY KEY (`Id`)
12 ) ENGINE=InnoDB DEFAULT CHARSET=latin1

1 CREATE TABLE `votes` (
2   `Id` int(11) NOT NULL,
3   `PostId` int(11) NOT NULL,
4   `UserId` int(11) DEFAULT NULL,
5   `BountyAmount` int(11) DEFAULT NULL,
6   `VoteTypeId` int(11) NOT NULL,
7   `CreationDate` datetime NOT NULL,
8   PRIMARY KEY (`Id`)
9 ) ENGINE=InnoDB DEFAULT CHARSET=latin1

```

The following SQL query will find the details of users who added my StackOverflow questions to their favorites list. For the sake of this example, my user id is 12345678.

```

1 SELECT
2   v.UserId,
3   COUNT(*) AS FavoriteCount
4 FROM
5   Votes v
6   JOIN Posts p ON p.id = v.PostId
7 WHERE
8   p.OwnerUserId = 12345678
9   AND v.VoteTypeId = 5 -- (Favorites vote)
10 GROUP BY
11   v.UserId
12 ORDER BY
13   FavoriteCount DESC
14 LIMIT
15   100;

```

The original query's execution duration is very long. I stopped waiting and cancelled the execution after more than a minute has passed.

Action	Message	Duration / Fetch
SELECT v.UserId, COUNT(*) AS FavoriteCount FROM Votes v JOIN Posts p ON p.id = v.PostId WH...	Error Code: 1317. Query execution wa...	74.116 sec
INTERRUPT	OK - Query cancelled	0.000 sec

# Explaining the original EXPLAIN

This is the original EXPLAIN plan for this query:

id	select_type	table	part	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	v	NULL	ALL	NULL	NULL	NULL	NULL	145045879	10.00	Using where; Using temporary; Using filesort
1	SIMPLE	p	NULL	eq_ref	PRIMARY	PRIMARY	4	so.v.PostId	1	10.00	Using where

Before rushing to optimize the query, let's take a closer look at the output of the EXPLAIN command, to make sure we fully understand all aspects of it. The first thing we notice, is that it can include more than one row. The query we're analyzing involves two tables in the process, which are joined using an inner join. Each of these tables gets represented in a different row in the execution plan above. As an analogy to the coding world, you can look at the concept of an inner join as very similar to a nested loop. MySQL chooses the table it thinks will be best to start the journey with (the outer "loop") and then touches the next table using the values from the outer "loop".

Each of the rows in the EXPLAIN contains the following fields:

- **id** - In most cases, the id field will present a sequential number of the SELECT query this row belongs to. The query above contains no subqueries nor unions, so therefore the id for both rows is 1, as there is actually only 1 query.
- **select\_type** - The type of SELECT query. In our case, it's a simple query as it contains no subqueries or unions. In more complex cases, it will contain other types such as SUBQUERY (for subqueries), UNION (second or later statements in a union), DERIVED (a derived table) and others. More information about access\_types can be found in [MySQL's docs](#).
- **table** - the table name, or alias, this row refers to. In the screenshot above, you can see 'v' and 'p' mentioned, as those are the aliases defined for the tables *votes* and *posts*.

- **type** - defines how the tables are accessed / joined. The most popular access types you'll generally see are the following, sorted from the worst to the best: ALL, index, range, ref, eq\_ref, const, system. As you can see in the EXPLAIN, the table *votes* is the first table accessed, using the ALL access\_type, which means MySQL will scan the entire table, using no indexes, so it will go through over 14 million records. The *posts* table is then accessed using the eq\_ref access type. Other than the system and const types, eq\_ref is the best possible join type. The database will access one row from this table for each combination of rows from the previous tables.
- **possible\_keys** - The optional indexes MySQL can choose from, to look up for rows in the table. Some of the indexes in this list can be actually irrelevant, as a result of the execution order MySQL chose. In general, MySQL can use indexes to join tables. Said that, it won't use an index on the first table's join column, as it will go through all of its rows anyway (except rows filtered by the WHERE clause).
- **key** - This column indicates the actual index MySQL decided to use. It doesn't necessarily mean it will use the entire index, as it can choose to use only part of the index, from the left-most side of it.
- **key\_len** - This is one of the important columns in the explain output. It indicates the length of the key that MySQL decided to use, in bytes. In the EXPLAIN output above, MySQL uses the entire PRIMARY index (4 bytes). We know that because the only column in the PRIMARY index is *Id*, which is defined as an INT => 4 bytes. Unfortunately, there is no easier way to figure out which part of the index is used by MySQL, other than aggregating the length of all columns in the index and comparing that to the key\_len value.
- **rows** - Indicates the amount of number of rows MySQL believes it must examine from this table, to execute the query. This is only an estimation. Usually, high row counts mean there is room for query optimization.
- **filtered** - The amount of rows unfiltered by the conditions in the WHERE clause. These rows will be joined to the table in the next row of the EXPLAIN plan. As mentioned previously, this is a guesstimate as well, so MySQL can be wrong with this estimation.

- **extra** - Contains more information about the query processing. Let's look into the extras for our query:
  - using where - The WHERE clause is used to restrict which rows are fetched from the current table (*votes*) and matched with the next table (*posts*).
  - using temporary - As part of the query processing, MySQL has to create a temporary table, which in many cases can result in a performance penalty. In most cases, it will indicate that one of the ORDER BY or GROUP BY clauses is executed without using an index. It can also happen if the GROUP BY and ORDER BY clauses include different columns (or in different order).
  - using filesort - MySQL is forced to perform another pass on the results of the query to sort them. In many cases, this can also result in a performance penalty.

## Optimizing a slow query using MySQL's EXPLAIN

---

What can we learn from this specific EXPLAIN plan?

1. MySQL chooses to start with the *votes* table. The EXPLAIN output shows that it has to go through 145,045,878 rows, which is about all the rows in the table. That's nasty.
2. The Extra column indicates that MySQL tries to reduce the amount of rows it inspects using the WHERE clause, but it estimates to reduce those to 10%, which is still bad, as 10% = 14 million rows. A possible conclusion from this information is that the condition *v.VoteTypeId* = 5 isn't selective enough, so therefore millions of rows will be joined to the next table.
3. Looking at the WHERE clause, we can see there is another condition, *p.OwnerUserId* = 12345678, which looks very selective and should drastically reduce the amount of rows to inspect. The *posts* table contains ~40 million

records, while only 57 rows are returned when applying the condition *p.OwnerUserId = 12345678*, which means that it's very selective. In this case, it would be best if MySQL would have started the execution using the *posts* table, to take advantage of this selective condition. Later in this post, we'll see which change will get MySQL to choose that order.

- Looking at the *possible\_keys* values for both tables, we can see that MySQL doesn't have a lot of optional indexes to choose from. More precisely, there is no index in the *possible\_keys* that contains the columns mentioned in the WHERE clause.

Therefore, we'll add the following two indexes. Each index starts with the column mentioned in the WHERE clause. The index for the *votes* table also includes the joined column.

```
1 ALTER TABLE `Posts` ADD INDEX `posts_idx_owneruserid` (`Owner
2 ALTER TABLE `Votes` ADD INDEX `votes_idx_votetypeid_postid` (
```

## The optimized query's EXPLAIN output

This is the new EXPLAIN output after adding the indexes:

id	select_type	table	partition	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	NULL	ref	PRIMARY,posts_idx_owneruserid	posts_idx_owneruserid	5	const	57	100.00	Using index; Using temporary; Using filesort
1	SIMPLE	v	NULL	ref	votes_idx_votetypeid_postid	votes_idx_votetypeid_postid	8	const,p.o.Id	2	100.00	NULL

What changed?

- The first change we see is that MySQL chose to start with the *posts* table this time (hurray!). It uses the new index to filter out the rows and estimates to filter all but 57 records, which are then joined to the second table, *votes*.
- The second change we see, by looking at the *key* column, is that indexes are used for lookups and filtering in both tables.

3. By looking at the `key_len` column, we can see that the composite index for the `votes` table is used in full - 8 bytes, which covers both the `VoteTypeId` and `PostId` columns.
4. The last important change we see is the amount of rows MySQL estimates it needs to inspect in order to run evaluate the query. It estimates it needs to inspect  $57 * 2 = 114$  rows, which is great, comparing to the millions of records in the original execution path.

So looking at the execution duration now, we can see a drastic improvement, from a very slow query which never actually returned, to only 0.063 seconds:

#	Time	Action	Message	Duration / Fetch
1	19:33:32	SELECT	v.UserId, COUNT(*) AS FavoriteCount FROM Votes v JOIN Posts p ON p.id = v.PostId WHERE p.Ow... 36 row(s) returned	0.063 sec / 0.000 sec

## Conclusions

Summarizing the most important aspects of reading an EXPLAIN and optimizing this simple query:

1. Run the EXPLAIN command to inspect the planned execution path for your SQL query.
2. Look at the tables order MySQL chose for the execution. Does it makes sense? If not, ask yourself why did MySQL get it wrong, what's missing?
3. Find the conditions in the WHERE clause which are the most selective ones and make sure you create the optimal indexes to include them. You can read more information on [how to create the optimal indexes here](#).
4. Look for the places MySQL doesn't use an index for look-ups and filtering, as those may be the weak spots.
5. Look for the rows where MySQL shows a very high estimation of rows it needs to inspect to evaluate the query.

In our next post for this series, we'll analyze a more complex query, looking into more insights the EXPLAIN can provide for other query structures such as sub-queries.

©2019 EVERSQL.COM