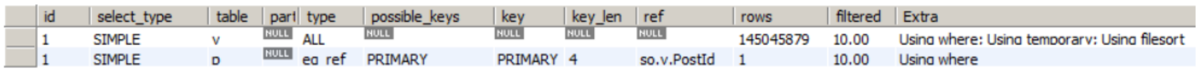
Explaining the original EXPLAIN

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.

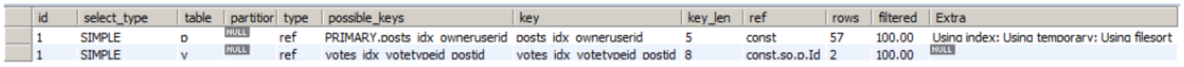
Optimizing a slow query using MySQL's EXPLAIN

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.

```
ALTER TABLE `Posts` ADD INDEX `posts_idx_owneruserid` (`OwnerUserId`);
ALTER TABLE `Votes` ADD INDEX `votes_idx_votetypeid_postid` (`VoteTypeId`,`PostId`);
```

The optimized query's EXPLAIN output

This is the new EXPLAIN output after adding the indexes:

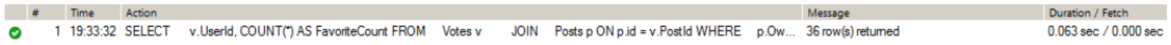


id	select_type	table	partitior	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	NULL	ref	PRIMARY,posts_idx_owneruserid	posts_idx_owneruserid	5	const	57	100.00	Using index; Using temporary; Using filesort
1	SIMPLE	v	NULL	ref	votes_idx_votetypeid_postid	votes_idx_votetypeid_postid	8	const,so.p.Id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

What changed?

1. 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.*
2. 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