Using EXPLAIN to Write Better MySQL Queries

When you issue a query, the MySQL Query Optimizer tries to devise an optimal plan for query execution. You can see information about the plan by prefixing the query with `EXPLAIN`. `EXPLAIN` is one of the most powerful tools at your disposal for understanding and optimizing troublesome MySQL queries, but it's a sad fact that many developers rarely make use of it. In this article you'll learn what the output of `EXPLAIN` can be and how to use it to optimize your schema and queries.

Understanding EXPLAIN's Output

Using `EXPLAIN` is as simple as pre-pending it before the `SELECT` queries. Let's analyze the output of a simple query to familiarize yourself with the columns returned by the command.

```
EXPLAIN SELECT   categoriesG
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: categories
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
```

```
         rows: 4
        Extra:
1 row in set (0.00 sec)
```

It may not seem like it, but there's a lot of information packed into those 10 columns! The columns returned by the query are:

- `id` – a sequential identifier for each `SELECT` within the query (for when you have nested subqueries)
- `select_type` – the type of `SELECT` query. Possible values are:
  - `SIMPLE` – the query is a simple `SELECT` query without any subqueries or `UNION`s
  - PRIMARY – the `SELECT` is in the outermost query in a `JOIN`
  - DERIVED – the `SELECT` is part of a subquery within a `FROM` clause
  - SUBQUERY – the first `SELECT` in a subquery
  - DEPENDENT SUBQUERY – a subquery which is dependent upon on outer query
  - UNCACHEABLE SUBQUERY – a subquery which is not cacheable (there are certain conditions for a query to be cacheable)
  - UNION – the `SELECT` is the second or later statement of a UNION
  - DEPENDENT UNION – the second or later `SELECT` of a `UNION` is dependent on an outer query
  - UNION RESULT – the `SELECT` is a result of a `UNION`

- `table` – the table referred to by the row
- `type` – how MySQL joins the tables used. This is one of the most insightful fields in the output because it can indicate missing indexes or how the query is written should be reconsidered. Possible values are:
  - system – the table has only zero or one row
  - const – the table has only one matching row which is indexed. This is the fastest type of join because the table only has to be read once and the column's value can be treated as a constant when joining other tables.
  - eq_ref – all parts of an index are used by the join and the index is `PRIMARY KEY` or `UNIQUE NOT NULL`. This is the next best possible join type.
  - ref – all of the matching rows of an indexed column are read for each combination of rows from the previous table. This type of join appears for indexed columns compared using `=` or <=> operators.
  - fulltext – the join uses the table's `FULLTEXT` index.

  - ref_or_null – this is the same as ref but also contains rows with a null value for the column.
  - index_merge – the join uses a list of indexes to produce the result set. The key column of `EXPLAIN`'s output will contain the keys used.
  - unique_subquery – an `IN` subquery returns only one result from the table and makes use of the primary key.
  - index_subquery – the same as unique_subquery but returns more than one result row.
  - range – an index is used to find matching rows in a specific range, typically when the key column is compared to a constant using operators like `BETWEEN`, `IN`, `>`, `>=`, etc.
  - index – the entire index tree is scanned to find matching rows.
  - all – the entire table is scanned to find matching rows for the join. This is the worst join type and usually indicates the lack of appropriate indexes on the table.

- `possible_keys` – shows the keys that can be used by MySQL to find rows from the table, though they may or may not be used in practice. In fact, this column can often help in optimizing queries since if the column is NULL, it indicates no relevant indexes could be found.
- `key` – indicates the actual index used by MySQL. This column may contain an index that is not listed in the `possible_key` column. MySQL optimizer always look for an optimal key that can be used for the query. While joining many tables, it may figure out some other keys which is not listed in `possible_key` but are more optimal.
- `key_len` – indicates the length of the index the Query Optimizer chose to use. For example, a `key_len` value of 4 means it requires memory to store four characters. Check out MySQL's data type storage requirements[1] to know more about this.
- `ref` – Shows the columns or constants that are compared to the index named in the key column. MySQL will either pick a constant value to be compared or a column itself based on the query execution plan. You can see this in the example given below.
- `rows` – lists the number of records that were examined to produce the output. This Is another important column worth focusing on optimizing queries, especially for queries that use `JOIN` and subqueries.
- `Extra` – contains additional information regarding the query execution plan. Values such as "Using temporary", "Using filesort", etc. in this column may indicate a troublesome query. For a complete list of possible values and their meaning, check out the MySQL documentation.

You can also add the keyword `EXTENDED` after `EXPLAIN` in your query and MySQL will show you additional information about the way it executes the query. To see the information, follow your `EXPLAIN` query with `SHOW WARNINGS`. This is mostly useful for seeing the query that is executed after any transformations have been made by the Query Optimizer.

```
EXPLAIN EXTENDED SELECT CityName  City
 Country  CityCountryCode  CountryCode
WHERE CityCountryCode  'IND'  CountryContinent  'Asia'G
********************* 1. row *********************
           id: 1
  select_type: SIMPLE
        table: Country
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 3
          ref: const
         rows: 1
     filtered: 100.00
        Extra:
********************* 2. row *********************
           id: 1
  select_type: SIMPLE
        table: City
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
```

```
         rows: 4079
     filtered: 100.00
        Extra: Using where
2 rows in set, 1 warning (0.00 sec)
```

```
 WARNINGSG
********************** 1. row **********************
  Level: Note
   Code: 1003
Message: select `World`.`City`.`Name` AS `Name` from `World`.`City` join `World`.`Country` where ((`World`.`City`.`CountryCode` = 'IND'))
1 row in set (0.00 sec)
```

Troubleshooting Performance with EXPLAIN

Now let's take a look at how we can optimize a poorly performing query by analyzing the output of `EXPLAIN`. When dealing with a real-world application there'll undoubtedly be a number of tables with many relations between them, but sometimes it's hard to anticipate the most optimal way to write a query.

Here I've created a sample database for an e-commerce application which does not have any indexes or primary keys, and will demonstrate the impact of such a bad design by writing a pretty awful query. You can download the schema sample[2] from GitHub.

```
EXPLAIN SELECT
orderdetails
INNER  orders o  orderNumber  oorderNumber
INNER  products p  pproductCode  productCode
INNER  productlines l  pproductLine  lproductLine
```

```
INNER  customers   customerNumber  ocustomerNumber
WHERE oorderNumber  10101G
********************* 1. row *********************
           id: 1
  select_type: SIMPLE
        table: l
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 7
        Extra:
********************* 2. row *********************
           id: 1
  select_type: SIMPLE
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 110
        Extra: Using where; Using join buffer
********************* 3. row *********************
           id: 1
  select_type: SIMPLE
        table: c
         type: ALL
```

```
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 122
        Extra: Using join buffer
********************** 4. row **********************
           id: 1
  select_type: SIMPLE
        table: o
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 326
        Extra: Using where; Using join buffer
********************** 5. row **********************
           id: 1
  select_type: SIMPLE
        table: d
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2996
        Extra: Using where; Using join buffer
5 rows in set (0.00 sec)
```

If you look at the above result, you can see all of the symptoms of a bad query. But even if I wrote a better query, the results would still be the same since there are no indexes. The join type is shown as "ALL" (which is the worst), which means MySQL was unable to identify any keys that can be used in the join and hence the `possible_keys` and `key` columns are null. Most importantly, the `rows` column shows MySQL scans all of the records of each table for query. That means for executing the query, it will scans 7 × 110 × 122 × 326 × 2996 = 91,750,822,240 records to find the four matching results. That's really horrible, and it will only increase exponentially as the database grows.

Now lets add some obvious indexes, such as primary keys for each table, and execute the query once again. As a general rule of thumb, you can look at the columns used in the `JOIN` clauses of the query as good candidates for keys because MySQL will always scan those columns to find matching records.

```
ALTER TABLE customers
     PRIMARY  customerNumber
ALTER TABLE employees
     PRIMARY  employeeNumber
ALTER TABLE offices
     PRIMARY  officeCode
ALTER TABLE orderdetails
     PRIMARY  orderNumber productCode
ALTER TABLE orders
     PRIMARY  orderNumber
      customerNumber
ALTER TABLE payments
     PRIMARY  customerNumber checkNumber
ALTER TABLE productlines
     PRIMARY  productLine
```

```
ALTER TABLE products
     PRIMARY  productCode
      buyPrice
      productLine
ALTER TABLE productvariants
     PRIMARY  variantId
      buyPrice
      productCode
```

Let's re-run the same query again after adding the indexes and the result should look like this:

```
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: o
         type: const
possible_keys: PRIMARY,customerNumber
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
********************** 2. row **********************
           id: 1
  select_type: SIMPLE
        table: c
         type: const
possible_keys: PRIMARY
          key: PRIMARY
```

```
      key_len: 4
          ref: const
         rows: 1
        Extra:
********************** 3. row **********************
           id: 1
  select_type: SIMPLE
        table: d
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 4
        Extra:
********************** 4. row **********************
           id: 1
  select_type: SIMPLE
        table: p
         type: eq_ref
possible_keys: PRIMARY,productLine
          key: PRIMARY
      key_len: 17
          ref: classicmodels.d.productCode
         rows: 1
        Extra:
********************** 5. row **********************
           id: 1
  select_type: SIMPLE
```

```
        table: l
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 52
          ref: classicmodels.p.productLine
         rows: 1
        Extra:
5 rows in set (0.00 sec)
```

After adding indexes, the number of records scanned has been brought down to 1 × 1 × 4 × 1 × 1 = 4. That means for each record with `orderNumber` 10101 in the `orderdetails` table, MySQL was able to directly find the matching record in all other tables using the indexes and didn't have to resort to scanning the entire table.

In the first row's output you can see the join type used is "const," which is the fastest join type for a table with more than one record. MySQL was able to use `PRIMARY` key as the index. The ref column shows "const," which is nothing but the value 10101 used in the query's `WHERE` clause.

Let's take a look at one more example query. Here we'll basically take the union of two tables, `products` and `productvariants`, each joined with `productline`. `productvariants` table consists of different product variants with `productCode` as reference keys and their prices.

```
EXPLAIN SELECT
SELECT pproductName pproductCode pbuyPrice lproductLine pstatus lstatus  lineStatus
products p
INNER  productlines l  pproductLine  lproductLine
UNION
SELECT vvariantName  productName vproductCode pbuyPrice lproductLine pstatus lstatus  lineStatus  productvariants v
```

```
INNER  products p  pproductCode  vproductCode
INNER  productlines l  pproductLine  lproductLine
 products
WHERE status  'Active'  lineStatus  'Active'  buyPrice BETWEEN   50G
********************** 1. row **********************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 219
        Extra: Using where
********************** 2. row **********************
           id: 2
  select_type: DERIVED
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 110
        Extra:
********************** 3. row **********************
           id: 2
```

```
  select_type: DERIVED
        table: l
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 52
          ref: classicmodels.p.productLine
         rows: 1
        Extra:
********************** 4. row **********************
           id: 3
  select_type: UNION
        table: v
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 109
        Extra:
********************** 5. row **********************
           id: 3
  select_type: UNION
        table: p
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 17
          ref: classicmodels.v.productCode
```

```
         rows: 1
        Extra:
********************** 6. row **********************
           id: 3
  select_type: UNION
        table: l
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 52
          ref: classicmodels.p.productLine
         rows: 1
        Extra:
********************** 7. row **********************
           id: NULL
  select_type: UNION RESULT
        table: <union2,3>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra:
7 rows in set (0.01 sec)
```

You can see a number of problem in this query. It scans all records in the `products` and `productvariants` tables. As there are no indexes on these tables for the `productLine` and `buyPrice` columns, the output's `possible_keys` and `key` columns show null. The status of `products` and `productlines` is checked after the `UNION`, so moving them inside the `UNION` will reduce the number of records. Let's add some additional indexes and rewrite the query.

```
CREATE INDEX idx_buyPrice  productsbuyPrice

CREATE INDEX idx_buyPrice  productvariantsbuyPrice

CREATE INDEX idx_productCode  productvariantsproductCode

CREATE INDEX idx_productLine  productsproductLine

EXPLAIN SELECT

SELECT pproductName pproductCode pbuyPrice lproductLine pstatus lstatus  lineStatus  products p

INNER  productlines  l  pproductLine  lproductLine  pstatus  'Active'  lstatus  'Active'

WHERE buyPrice BETWEEN

UNION

SELECT vvariantName  productName vproductCode pbuyPrice lproductLine pstatus lstatus  productvariants v

INNER  products p  pproductCode  vproductCode  pstatus  'Active'

INNER  productlines l  pproductLine  lproductLine  lstatus  'Active'

WHERE

vbuyPrice BETWEEN

 productG
********************* 1. row *********************
          id: 1
  select_type: PRIMARY
        table: <derived2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 12
        Extra:
```

```
********************** 2. row **********************
           id: 2
  select_type: DERIVED
        table: p
         type: range
possible_keys: idx_buyPrice,idx_productLine
          key: idx_buyPrice
      key_len: 8
          ref: NULL
         rows: 23
        Extra: Using where
********************** 3. row **********************
           id: 2
  select_type: DERIVED
        table: l
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 52
          ref: classicmodels.p.productLine
         rows: 1
        Extra: Using where
********************** 4. row **********************
           id: 3
  select_type: UNION
        table: v
         type: range
possible_keys: idx_buyPrice,idx_productCode
          key: idx_buyPrice
```

```
      key_len: 9
          ref: NULL
         rows: 1
        Extra: Using where
********************** 5. row **********************
           id: 3
  select_type: UNION
        table: p
         type: eq_ref
possible_keys: PRIMARY,idx_productLine
          key: PRIMARY
      key_len: 17
          ref: classicmodels.v.productCode
         rows: 1
        Extra: Using where
********************** 6. row **********************
           id: 3
  select_type: UNION
        table: l
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 52
          ref: classicmodels.p.productLine
         rows: 1
        Extra: Using where
********************** 7. row **********************
           id: NULL
  select_type: UNION RESULT
```

```
        table: <union2,3>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra:
7 rows in set (0.01 sec)
```

As you can see in the result, now the number of approximate rows scanned is significantly reduced from 2,625,810 (219 × 110 × 109) to 276 (12 × 23), which is a huge performance gain. If you try the same query, without the previous re-arrangements, simply after adding the indexes, you wouldn't see much of a reduction. MySQL isn't able to make use of the indexes since it has the `WHERE` clause in the derived result. After moving those conditions inside the `UNION`, it is able to make use of the indexes. This means just adding an index isn't always enough; MySQL won't be able to use it unless you write your queries accordingly.

Summary

In this article I discussed the MySQL `EXPLAIN` keyword, what its output means, and how you can use its output to construct better queries. In the real world, it can be more useful than the scenarios demonstrated here. More often than not, you'll be joining a number of tables together and using complex `WHERE` clauses. Simply added indexes on on a few columns may not always help, and then it's time to take a closer look at your queries themselves.

Image via Efman[3] / Shutterstock[4]

Links

1. vhttp://dev.mysql.com/doc/refman/5.0/en/storage-requirements.html
2. https://github.com/phpmasterdotcom/UsingExplainToWriteBetterMySQLQueries
3. http://www.shutterstock.com/gallery-442516p1.html
4. http://www.shutterstock.com/