Optimizing Queries with EXPLAIN

Sheeri Cabral
Senior Database Administrator

Twitter: @sheeri

What is EXPLAIN?

SQL Extension

- Just put it at the beginning of your statement

Can also use DESC or DESCRIBE

- e.g. DESC SELECT COUNT(*) FROM mysql.user;

Used to be SELECT only (5.5 and lower)

5.6 and higher - SELECT, INSERT, UPDATE, DELETE, REPLACE

What EXPLAIN shows

Lots of information

- # and name of tables, views
- How tables are joined
- How data are looked up
- Subqueries, unions, sorts
- WHERE, DISTINCT
- Indexes considered
- Indexes used
- Length of indexes used
- Approx # rows looked at
- And more….

Metadata

To choose an index, the MySQL optimizer uses metadata

- Cardinality (# elements in the set). Lower = more duplicates.
  - A yes/no field has a cardinality of 2
  - A unique field that has 100 rows populated has a cardinality of 100.
- Total # rows

InnoDB has approximate statistics

- ANALYZE TABLE does a few dives into the data
- Configurable by innodb_stats_sample_pages (5.5), innodb_stats_transient_sample_pages (5.7)
- Fast (locking), but not accurate
- OPTIMIZE TABLE slow, locking, but accurate

EXPLAIN OUTPUT

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
```

id

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
```

Sequential identifier
One per table, subquery, derived table
Views are virtual, no rows returned
Underlying tables are represented

select_type

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
```

If the table is used in a subquery/union, or SIMPLE
PRIMARY = table is the outermost in a union or subquery
UNION, SUBQUERY – table is the 2nd or later in a union/subquery
DEPENDENT UNION, SUBQUERY – 2nd or later, dependent on outer query
Re-evaluated once for unique variables in the outer query
UNION RESULT – final row of EXPLAIN in a UNION
DERIVED – subquery in the FROM clause, aka derived table
UNCACHEABLE SUBQUERY, UNION – subquery (even in UNION) is re-evaluated for every outer row

Note: DEPENDENT is better than UNCACHEABLE; look for small rows examined and returned; a JOIN is best if possible

select_type in UNION queries

```
mysql> EXPLAIN SELECT first_name FROM staff UNION SELECT first_name FROM
customer\G
********** 1. row **********
           id: 1
  select_type: PRIMARY
        table: staff
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra:
********** 2. row **********
           id: 2
  select_type: UNION
        table: customer
         type: ALL
possible_keys: NULL
          key: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 541
     filtered: 100.00
        Extra:
********** 3. row **********
           id: NULL
  select_type: UNION RESULT
        table: <union1,2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra:
3 rows in set (0.00 sec)
```

Table name

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
```

The name of the table

NULL if no table is used

```
EXPLAIN SELECT 1+1\G
```

NULL if query is impossible

```
EXPLAIN SELECT return_date FROM rental WHERE rental_id=0\G
```

Why is that query impossible, how does the optimizer know?

Aliases dominate here

EXPLAIN “SELECT …. FROM rental as r” shows “table: r”

Aliases

```
mysql> EXPLAIN SELECT first_name FROM staff AS s UNION SELECT first_name
FROM customer\G
********** 1. row **********
           id: 1
  select_type: PRIMARY
        table: s
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra:
********** 2. row **********
           id: 2
  select_type: UNION
        table: customer
         type: ALL
possible_keys: NULL
          key: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 541
     filtered: 100.00
        Extra:
********** 3. row **********
           id: NULL
  select_type: UNION RESULT
        table: <union1,2>
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra:
3 rows in set (0.00 sec)
```

Partitions

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
```

Which partitions serve matching records
Only if EXPLAIN PARTITIONS is used
NULL for non-partitioned tables

type

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
```

“type” is one of the fun entries
It shows the data access strategy

Data Access Strategy

Get this as good as possible

This list is in terms of worst to best.

ALL – full table scan. DOES NOT USE AN INDEX. (#12)

- Might be OK if there are very few rows in the table

index – full index scan (#11)

- If you must scan all the data, consider a covering index.

range – partial index scan (#10)

- < <= > >=
- IS NULL, BETWEEN, IN
- Some OR statements whose parts use the same index
  - WHERE customer_id=1 OR customer_id=2

Range Query (Data Access Strategy #10)

```
mysql> EXPLAIN SELECT rental_id
    -> FROM rental
    -> WHERE rental_date BETWEEN
    -> '2006-02-14 00:00:00' and '2006-02-14 23:59:59'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 5
          ref: NULL
         rows: 182
     filtered: 100.00
        Extra: Using where; Using index
```

What data access strategy will be used, and why?

```
mysql> EXPLAIN SELECT rental_id
    -> FROM rental
    -> WHERE date(rental_date)='2006-02-14'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: index          ←Data Access Strategy #11. Worse than range
possible_keys: NULL
          key: rental_date
      key_len: 10
          ref: NULL
         rows: 16005
     filtered: 100.00
        Extra: Using where; Using index
```

Data Access Strategies

index_subquery (#9)

- Subquery using non-unique index of one table

unique_subquery (#8)

- Subquery using unique index of one table (PRIMARY, UNIQUE)

Data Access Strategies

index_merge (#7)

- Use more than one index
- Not used with fulltext indexes
- Extra field shows more information
  - intersection, union, sort_union algorithms
  - Index_merge_optimization in MySQL manual
    - **https://dev.mysql.com/doc/refman/5.7/en/index-merge-optimization.html**

- Some OR queries

Rental Table Has Indexes on customer_id and rental_date

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_date
    -> BETWEEN '2006-02-14 00:00:00' AND '2006-02-14 23:59:59’
    -> OR customer_id = 5\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: index_merge
possible_keys: rental_date,idx_fk_customer_id
          key: rental_date,idx_fk_customer_id
      key_len: 5,2
          ref: NULL
         rows: 220
     filtered: 100.00
        Extra: Using sort_union(rental_date,idx_fk_customer_id); Using
where
```

But what if there was an index that had both?

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_date
    -> BETWEEN '2006-02-14 00:00:00' AND '2006-02-14 23:59:59’
    -> OR customer_id = 5\G
```

e.g. index on (customer_id, rental_date) instead of just customer_id.

Data Access Strategies

ref_or_null (#6)

- Joining or looking up non-unique index values
- JOIN can use key prefix or non-unique index
- Indexed fields compared with =, !=, <=>
- If the fields involved are defined as nullable, an extra pass is done to search for NULL values
- Try to use NOT NULL as a default

Data Access Strategy #5

fulltext

```
mysql> EXPLAIN SELECT film_id, title FROM film_text
WHERE MATCH (title,description) AGAINST ('storm')\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: film_text
         type: fulltext
possible_keys: idx_title_description
          key: idx_title_description
      key_len: 0
          ref:
         rows: 1
        Extra: Using where
```

Data Access Strategies

ref (#4)

- Like ref_or_null (#6) but without the extra pass to look for NULL values
- Joining or looking up non-unique index values
- JOIN can use key prefix or non-unique index
- Indexed fields compared with =, !=, <=>
- The best you can get unless you can use a PRIMARY or UNIQUE index

Which Data Access Strategies?

```
mysql> EXPLAIN SELECT c.first_name, c.last_name, p.amount
FROM customer c INNER JOIN payment p USING (customer_id)\G

******* 1. row *******
           id: 1
  select_type: SIMPLE
        table: c
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 599
     filtered: 100.00
        Extra: NULL

******* 2. row *******
           id: 1
  select_type: SIMPLE
        table: p
   partitions: NULL
         type: ref
possible_keys: idx_fk_customer_id
          key: idx_fk_customer_id
      key_len: 2
          ref: sakila.c.customer_id
         rows: 26
     filtered: 100.00
        Extra: NULL
```

Data Access Strategies

eq_ref (#3)

- Like ref (#4) but on unique index values
- Joining on unique index values
- JOIN can use key prefix or unique index
- Indexed fields compared with =
- Pretty darn good

Which Data Access Strategies For Each Table?

Non-unique index on p.rental_id, p.customer_id, unique on c.customer_id

```
mysql> EXPLAIN SELECT first_name,last_name FROM customer c
INNER JOIN payment p USING (customer_id) WHERE rental_id=12345\G
************* 1. row *************
           id: 1
  select_type: SIMPLE
        table: p
   partitions: NULL
         type: ref
possible_keys: idx_fk_customer_id,
               fk_payment_rental
          key: fk_payment_rental
      key_len: 5
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
```

```
************* 2. row *************
           id: 1
  select_type: SIMPLE
        table: c
   partitions: NULL
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.p.customer_id
         rows: 1
     filtered: 100.00
        Extra: NULL
```

Data Access Strategies

But wait···.there's more!

const (#2)

- Uses a primary or unique key, will return at most 1 value

```
EXPLAIN SELECT return_date FROM rental AS r WHERE rental_id = 13534\G
```

system (#1)

- Same as "const" but the table only has 1 row (system table)

NULL (#0)

- There is no data to access! (same as table is null)
- Optimizer may access metadata

EXPLAIN Data Access Strategies

system - Table has one value (a system table)

const - Exactly one matching row (PRIMARY or UNIQUE keys)

eq_ref - One row from this table matches each combination of rows from previous tables.

ref - All matching rows from this table for each combination of rows from previous tables.

fulltext - FULLTEXT index

ref_or_null - ref with an additional pass for NULL values

index_merge - Index merge optimization used

unique_subquery - Optimized subquery handling in queries of the form value IN (SELECT …) using PRIMARY or UNIQUE keys

index_subquery - Like unique_subquery but with non-unique indexes range KEY is compared against =, <>, >, >=, <=, IS NULL, <=>, BETWEEN, or IN()

index - Full index scan

ALL - Full table Scan

The rest of the EXPLAIN output

possible_keys

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
possible_keys: PRIMARY
```

If MySQL did not consider the index you think it should, there is a reason:

- Query sorting
- Index order
- Other filters
- Data size
- Amount of data searched, if over about 15%

Amount of Data Searched

```
mysql> EXPLAIN SELECT * FROM rental WHERE rental_date BETWEEN
'2005-06-01' AND '2005-07-06'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: range
possible_keys: rental_date
          key: rental_date
      key_len: 5
          ref: NULL
         rows: 2338
     filtered: 100.00
        Extra: Using index condition
```

At what point does the data get so much that it’s faster to do a full table scan?

Answer: not as big as you think

```
Mysql> EXPLAIN SELECT * FROM rental WHERE rental_date BETWEEN
'2005-06-01' AND '2005-07-07'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: rental_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16067
     filtered: 17.69
        Extra: Using where
```

Remember this query and EXPLAIN plan?

```
mysql> EXPLAIN SELECT rental_id
    -> FROM rental
    -> WHERE date(rental_date)='2006-02-14'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: index
possible_keys: NULL
          key: rental_date
      key_len: 10
          ref: NULL
         rows: 16005
     filtered: 100.00
        Extra: Using where; Using index
```

Note that possible keys is NULL. That’s a BIG red flag.

It ends up using the rental_date index because it is a full index scan of that index.

key, key_len

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
```

Longer keys take a longer time to traverse and compare

A key_len shorter than the length of the key shows that only part of the key was used

- Index prefix
- One part of a compound index

ref

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
```

Ref is which column (or “const” for a number) is compared using the index from “key”

Remember this query and EXPLAIN plan?

```
mysql> EXPLAIN SELECT rental_id
    -> FROM rental
    -> WHERE date(rental_date)='2006-02-14'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: index
possible_keys: NULL
          key: rental_date
      key_len: 10
          ref: NULL
         rows: 16005
     filtered: 100.00
        Extra: Using where; Using index
```

Note that ref is NULL. This is a BIG clue that the optmizer has no value to compare to the index. That's why it does a full index scan.

rows and filtered

```
mysql> EXPLAIN SELECT return_date FROM rental WHERE rental_id=13534\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
     filtered: 100.00
```

Rows − how many rows the optimizer believes it needs to scan. Metadata is important!

LIMIT may change rows examined, but EXPLAIN still shows as if there is no LIMIT.

Filtered - % of **rows** that will be returned (cardinality!)

filtered

```
mysql> EXPLAIN SELECT * FROM rental WHERE return_date
BETWEEN '2006-02-14 00:00:00' and '2006-02-14 23:59:59'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16005
     filtered: 11.11
        Extra: Using where
```

Doing a full table scan of about 16,000 records, will return about 1,600 records.

Index cardinality

```
mysql> SELECT INDEX_NAME,SEQ_IN_INDEX,COLUMN_NAME,CARDINALITY
FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_SCHEMA='sakila' AND TABLE_NAME='rental';
+---------------------+--------------+---------------+-------------+
| INDEX_NAME          | SEQ_IN_INDEX | COLUMN_NAME   | CARDINALITY |
+---------------------+--------------+---------------+-------------+
| PRIMARY             |            1 | rental_id     |       16067 |
| rental_date         |            1 | rental_date   |       15815 |
| rental_date         |            2 | inventory_id  |       16044 |
| rental_date         |            3 | customer_id   |       16044 |
| idx_fk_inventory_id |            1 | inventory_id  |        4580 |
| idx_fk_customer_id  |            1 | customer_id   |         599 |
| idx_fk_staff_id     |            1 | staff_id      |           2 |
+---------------------+--------------+---------------+-------------+
7 rows in set (0.00 sec)
```

Extra

```
mysql> EXPLAIN SELECT * FROM rental WHERE return_date
BETWEEN '2006-02-14 00:00:00' and '2006-02-14 23:59:59'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: rental
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 16005
     filtered: 11.11
        Extra: Using where
```

Extra can be good or bad, depending. It is often self-explanatory, like “Using where”.

Extra

Using where − if you do not have this, make sure that is what you want.

Using index − only uses the index, does not go to the data. Covering index.

Using filesort − uses filesort ALGORITHM to do an extra pass through the data for sorting.

Distinct, **not exists** − stops at first row match

Temporary - intermediate temporary table used to store results.

Index_merge algorithm

All the values are in the MySQL manual, “EXPLAIN output” page.

EXPLAIN FORMAT=JSON

Json format

But also, more information!

LOTS more information…ALWAYS use it! (Except in MySQL 5.5)

EXPLAIN FORMAT=JSON

```
mysql> EXPLAIN FORMAT=JSON SELECT rental_id
    -> FROM rental
    -> WHERE date(rental_date)='2006-02-14'\G
************ 1. row ************
EXPLAIN: {
  "query_block":
 {    "select_id": 1,
    "cost_info": {
      "query_cost": "3310.40”
    },
    "table": {
      "table_name": "rental",
      "access_type": "index",
      "key": "rental_date",
      "used_key_parts": [
        "rental_date",
        "inventory_id",
        "customer_id”
      ],
      "key_length": "10",
      "rows_examined_per_scan": 16067,
      "rows_produced_per_join": 16067,
      "filtered": "100.00",
      "using_index": true,
      "cost_info": {
        "read_cost": "97.00",
        "eval_cost": "3213.40",
        "prefix_cost": "3310.40",
        "data_read_per_join": "502K”
      },
      "used_columns": [
        "rental_id",
        "rental_date”
      ],
      "attached_condition":
 "(cast(`sakila`.`rental`.`rental_date` as
date) = '2006-02-14')"    }  }}
```

Questions?
Feedback?
sheeri.com
@sheeri