# Practical 9: Query optimization

GAHAN SARAIYA (18MCEC10), RUSHI TRIVEDI(18MCEC08)

18mcec10@nirmauni.ac.in, 18mcec08@nirmauni.ac.in

## I. INTRODUCTION

Aim of this practical is to identify where default query processing can be optimized for cost or performance.

- Identify any scenario , where default query processing can be optimized for cost or performance.
- Specify your goal and type of query for which you are proposing query optimization.
- Implement and compare result with existing approach

## I. Purpose

If queries are bad, even the best-designed schema will not perform well. Hence queries needs to be well designed for achieving better performance. Query optimization, index optimization, and schema optimization go hand in hand.

## II. OPTIMIZE DATA ACCESS

The most basic cause a query does not perform well is because it's working with too much data. Some queries just have to analyze through a lot of data and can't be helped. That's unusual, though; most bad/time consuming queries can be altered to access less data. To analyze poorly performances query we need to identify:

- Whether program is retrieving more data than you need (accessing too many rows or columns)
- Whether if SQL server is analyzing the rows more than it needs.

## I. Querying Non Existing Data?

Some queries ask for more data than they need and then throw some of it away. This demands extra work of the SQL server, adds network overhead, and consumes memory and CPU resources on the application server.

### I.1 Fetching more rows than needed

One common mistake is assuming that SQL provides results on demand, rather than calculating and returning the full result set. We often see this in applications designed by people familiar with other database systems. These developers are used to techniques such as issuing a SELECT statement that returns many rows, then **fetching the first N rows**, and closing the result set. The general myth is that SQL will provide them with these first N rows and stop executing the query, but what SQL really does is **generate the complete result set**. The client library then fetches all the data and discards most of it.

The solution is to add a LIMIT clause to the query.

### I.2 Fetching all columns from a multi-table join

By Selecting only needed columns we can decrease data access require to perform query.

Unoptimized query:

```
SELECT * FROM moviedb.actor
INNER JOIN moviedb.film_actor USING(actor_id)
INNER JOIN moviedb.film USING(film_id)
WHERE moviedb.film.title = 'The Avengers';
```

Optimized query:

```
SELECT moviedb.actor.* FROM moviedb.actor
INNER JOIN moviedb.film_actor USING(actor_id)
INNER JOIN moviedb.film USING(film_id)
WHERE moviedb.film.title = 'The Avengers';
```

Some DBAs ban **SELECT *** universally because of this fact, and to reduce the risk of problems when someone alters the table's column list.

## II. Examining too much data?

Once you're sure your queries retrieve only the data you need, you can look for queries that examine too much data while generating results. In SQL, the simplest query cost metrics are:

- Execution time
- Number of rows examined
- Number of rows returned

None of these metrics is a perfect way to measure query cost, but they reflect roughly how much data must access internally to execute a query and translate approximately into how fast the query runs. All three metrics are logged in the slow query log, so looking at the slow query log is one of the best ways to find queries that examine too much data.

## III. Optimize Like Statements With Union Clause

Sometimes, you may want to run queries using the comparison operator 'or' on different fields or columns in a particular table. When the 'or' keyword is used too much in where clause, it might make the Query optimizer to incorrectly choose a full table scan to retrieve a record.

A union clause can make the query run faster especially if you have an index that can optimize one side of the query and a different index to optimize the other side.

consider a case where you are running the below query with the '$film_actor.first_name$' and '$film_actor.first_name$' indexed

```
SELECT * from moviedb
WHERE film_actor.first_name LIKE  'Robert%'  or film_actor.last_name LIKE 'Robert%' ;
```

The query above can run far much slower compared to the query written below which uses a union operator merge the results of 2 separate fast queries that takes advantage of the indexes.

```
SELECT from moviedb
WHERE film_actor.first_name like  'Robert%'
UNION ALL SELECT from moviedb
WHERE film_actor.last_name like  'Robert%' ;
```

## III. SUMMARY

Though fetching more data degrade performance; asking for more data than really needed is not always bad. In many cases it lets the developer use the same bit of code in more than one place. That's a reasonable consideration, as long as one is aware of what it costs in terms of performance.
It may also be useful to retrieve more data than you actually need if you use some type of caching in your application, or if you have another benefit in mind. Fetching and caching full objects may be preferable to running many separate queries that retrieve only parts of the object.

Detailed explanation of optimizing MYSQL queries can be found here and for PostgreSQL it can be found at this link.