Improving Query Performance Using Index

Puneet Kumar
DataOps Engineer
https://www.linkedin.com/in/puneetkumardataopsengineer/

Before You Begin

- This presentation is MySQL specific, but most of the concepts can also be applied to the other databases.
- The slides uses the employees sample database.
- Download: https://launchpad.net/test-db/employees-db-1/1.0.6/+download/employees_db-dump-files-1.0.5.tar.bz2
- How to install on your local machine?

  ```
  $> tar -xzvf employees_db-dump-files-1.0.5.tar.bz2
  $> mysql –user=root –password=yourmysqlpass -t < ./employees_db/employees.sql
  ```
- I have made some changes to this database for suitable examples.

Contents

- What affects Database performance
- What is Database Index
- Types Of Database Index
- Column Index
- Composite Index
- Covering Index
- Indexing Guidelines
- Conclusion
- Resources
- Bonus Slides

What Affects Database Performance?

- Hardware of the Database machine
- How you have configured your Database
- Physical Implementation of the Database
- Logical design of the Database
  1. How you have designed the Database schema.
  2. How you have designed & used Database Index.
  3. How you have used data types for the Columns.
- How SQL queries are written

What is Database Index?

- Is a Data Structure (most commonly a B-tree)
- Improves speed of data retrieval from Database Table.
- Used to quickly locate data without having to search every row in the table.
- Index stores the values from indexed column.
- Also store pointers to the corresponding rows in the table.
- Indexes don't come for free:
  1. Extra Space- Depends on size of the table and datatype of the column being indexed.
  2. INSERT, UPDATE and DELETE operation will be comparatively slower.

Types Of Database Index - I

- Clustered
    - ➔ Here leaf nodes, contain the actual data pages of the underlying table.
    - ➔ Only one clustered index can be created on a given database table
    - ➔ The physical order of the rows is same as the clustered index.

- Nonclustered
    - ➔ Here leaf nodes contains pointer to actual data row.
    - ➔ There can be more than one non-clustered index on a database table.
    - ➔ The physical order of the rows is not the same as the index order

Types Of Database Index - II

- FullText- FULLTEXT indexes have an “inverted index” design. Inverted indexes store a list of words, and for each word, a list of documents that the word appears in. Only useful for full text searches.
- Unique- Ensures that the index key contains no duplicate values
- Spatial- Provides the ability to perform certain operations more efficiently on spatial(geo) data.
- Filtered- An optimized nonclustered index. It uses a filter predicate to index a portion of rows in the table.

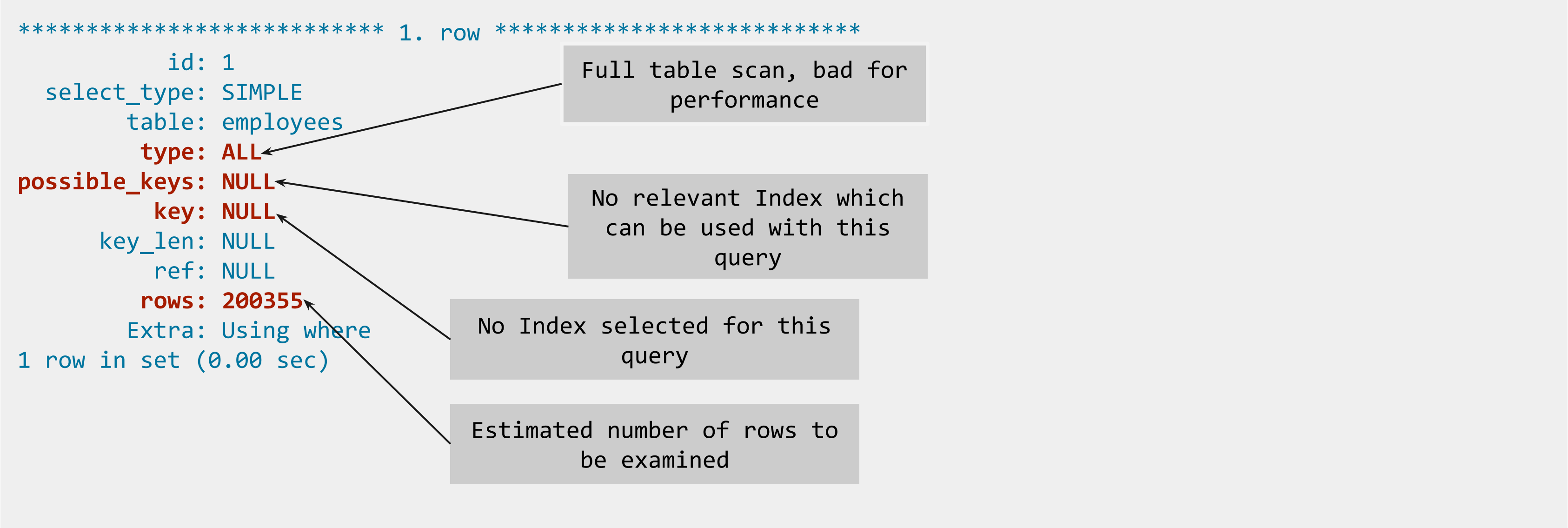
Column Index - I

- Use of indexes on the relevant columns is the best way to improve the performance of SELECT operations.
- The maximum number of indexes per table and the maximum index length is defined per storage engine.
- All storage engines support at least 16 indexes per table and a total index length of at least 256 bytes.
- Examples:-

Column Index - II

- Following query is to retrieve all the employees whose first_name is ‘Georgi’.

```
mysql> EXPLAIN SELECT * FROM employees WHERE first_name='Georgi' \G
```



```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200355
        Extra: Using where
1 row in set (0.00 sec)
```

Full table scan, bad for performance

No relevant Index which can be used with this query

No Index selected for this query

Estimated number of rows to be examined

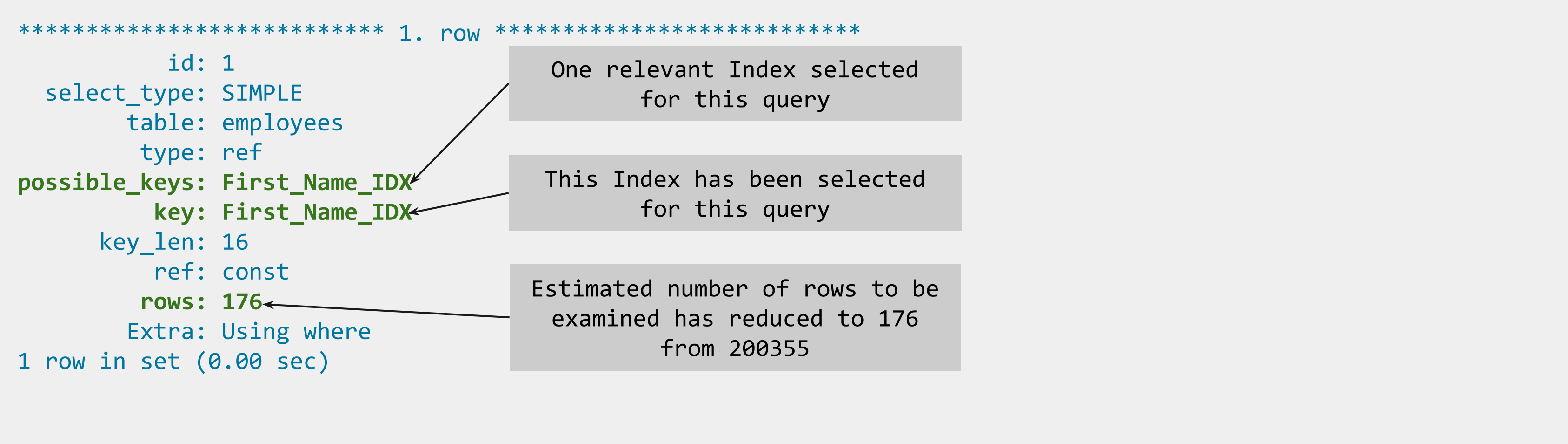
To know about EXPLAIN in MySQL, refer the Bonus Slides at the end.

Column Index - III

```
mysql> ALTER TABLE employees ADD INDEX First_Name_IDX(first_name);
```

```
Query OK, 0 rows affected (0.94 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT * FROM employees WHERE first_name='Georgi' \G
```



```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_Name_IDX
          key: First_Name_IDX
      key_len: 16
          ref: const
         rows: 176
        Extra: Using where
1 row in set (0.00 sec)
```

One relevant Index selected for this query

This Index has been selected for this query

Estimated number of rows to be examined has reduced to 176 from 200355

\G - Send command to mysql server and display result vertically.

Composite Index - I

- MySQL can create composite indexes (that is, indexes on multiple columns).
- Also called as Compound Index.
- An index may consist of up to 16 columns.
- MySQL can use multiple-column indexes for queries that test all the columns in the index, or queries that test just the first column, the first two columns, the first three columns, and so on.
- If you specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

Composite Index - II

- Examples:-

```
mysql> EXPLAIN SELECT * FROM employees WHERE first_name='Georgi' AND last_name='Facello' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_Name_IDX
          key: First_Name_IDX
      key_len: 16
          ref: const
         rows: 176
        Extra: Using where
1 row in set (0.00 sec)
```

Even though we have one extra AND condition when you compare with the last query, the number of rows to be examined remains the same. This is because MySQL has not found any better Index from the one, which has been used in the previous query.

Composite Index - III

```
mysql> ALTER TABLE employees ADD INDEX First_And_Last_Name_IDX(first_name,last_name);
```

```
Query OK, 0 rows affected (1.6 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT * FROM employees WHERE first_name='Georgi' AND last_name='Facello' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_Name_IDX,First_And_Last_Name_IDX
          key: First_And_Last_Name_IDX
      key_len: 34
          ref: const,const
         rows: 2
        Extra: Using where
1 row in set (0.00 sec)
```

Now, number of rows to be examined is 2, hence adding Index on both first_name and last_name will improve the performance.

Composite Index - IV

- If the table has a multiple-column index, any ***leftmost prefix*** of the index can be used by the optimizer to look up rows.
- For example, if you have a three-column index on (col1, col2, col3), you have indexed search capabilities on (col1), (col1, col2), and (col1, col2, col3).
- Following queries will form leftmost prefix, hence index can be used.

```
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;
```

- Following queries will not form leftmost prefix, hence index can not be used.

```
SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

Covering Index - I

- In most cases, an index is used to quickly locate the data record(s) from which the required data is read. A covering index is a special case where the index itself contains the required data field(s) and can return the data.
- Use Covering Index when you have high Cardinality (Number of unique values in the column).
- Covering indexes will improve performance because you don’t need to do any extra ***disk I/O*** to read the values as they are already there in the index.
- Can speed up data retrieval, but Index may itself grow large due to the additional keys, which will slow down data insertion & update.

Covering Index - II

- Examples:-

The following query is to retrieve all the employee’s emp_no and salary, whose salary is in between particular range in particular date range (from_date to_date).

```
mysql> EXPLAIN SELECT emp_no,salary FROM salaries WHERE salary between (30000 AND 50000) AND from_date IN
       ('1986-06-26','1994-06-24','1991-06-25') AND to_date IN ('1987-06-26','1994-06-24') \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: salaries
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1897749
        Extra: Using where
1 row in set (0.00 sec)
```

Huge number of rows to be examined. We can think of adding Index on salary, from_date and to_date column.

Covering Index - III

```
mysql> ALTER TABLE salaries ADD INDEX Salary_From_And_To_Date_IDX(salary,from_date,to_date);
```

```
Query OK, 0 rows affected (21.88 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT emp_no,salary FROM salaries WHERE salary between 30000 AND 50000 AND from_date IN
       ('1986-06-26','1994-06-24','1991-06-25') AND to_date IN ('1987-06-26','1994-06-24') \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: salaries
         type: range
possible_keys: Salary_From_And_To_Date_IDX
          key: Salary_From_And_To_Date_IDX
      key_len: 10
          ref: const,const
         rows: 34
        Extra: Using where; Using index
1 row in set (0.00 sec)
```

Now, number of rows to be examined is only 34. Without having the Salary_From_And_To_Date_IDX Index there was 1897749 rows to be examined.

Now this is interesting. We have new value ***Using index*** in ***Extra*** column. This means that, the column information is retrieved from the table using only information in the index tree without having to do an additional seek to read the actual row. (Covering Index)

When To Use Covering Index

- When you have large tables and there is similar query getting executed again and again.
- When we don't have too many columns to be selected from the table.
- When a lot of rows match the same key value, hence size of the index will not be an issue.
- When your application is read intensive- For example Bookmarking Application-In Bookmarking Application, you will not be creating/updating bookmarks frequently, rather you will be mostly reading bookmarks.

When you meet most of the above cases, you get best performance benefits from Covering Index.

When Not To Use Covering Index

- When you don't have to run the query using covering Index most frequently.
- On using Covering Index, the key length must not change significantly. Next slide explains how key length changes.
- When your application is write intensive- For example Activity History on your website. In this case there will be high write rate, and having covering index has overhead on INSERT, UPDATE & DELETE.

How Key Length Changes? - I

- Examples:-

```
mysql> EXPLAIN select * from employees where first_name = 'Georgi' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_And_Last_Name_IDX
          key: First_And_Last_Name_IDX
      key_len: 16
          ref: const
         rows: 176
        Extra: Using where
1 row in set (0.00 sec)
```

Indicates the length of the key in bytes that MySQL decided to use. But when you check first_name column data type in employees table, it varchar(14), then why key_len is 16.

Here is the rules to remember:-

1. If the field allows NULL, then it will have 1 more byte in the index to store the NULL flag.
2. If your field uses varchar, then it will have 2 more byte extra in the index for that field.

Since first_name column is using varchar(14) data type, so the key_len will be 14 + 2 = 16 bytes.

How Key Length Changes? - II

```
mysql> EXPLAIN select * from employees where first_name='Georgi' AND last_name='Facello' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_And_Last_Name_IDX
          key: First_And_Last_Name_IDX
      key_len: 34
          ref: const,const
         rows: 2
        Extra: Using where
1 row in set (0.00 sec)
```

In this case we have varchar(14) as datatype of first_name column and varchar(16) as datatype of last_name column. Hence the key_len can be computed as 14+2+16+2 = 34 bytes.

Note: Here the key length changes significantly, and when key length changes significantly we may not get the actual performance benefit from Covering Index. In such cases we should not create such Indexes.

Order of Columns in Index Matters - I

```
mysql> ALTER TABLE employees ADD INDEX First_And_Last_Name_IDX(first_name,last_name);
```

```
mysql> ALTER TABLE employees ADD INDEX DOB_First_And_Last_Name_IDX(birth_date,first_name,last_name);
```

```
Query OK, 0 rows affected (2.46 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT emp_no,birth_date FROM employees WHERE first_name='Georgi' AND last_name='Facello' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_And_Last_Name_IDX
          key: First_And_Last_Name_IDX
      key_len: 34
          ref: const,const
         rows: 2
        Extra: Using where
1 row in set (0.00 sec)
```

Even though we have Index on all the three columns birth_date, first_name and last_name. Not a Covering Index.

Order of Columns in Index Matters - II

```
mysql> ALTER TABLE employees ADD INDEX First_And_Last_Name_DOB_IDX(first_name,last_name,birth_date);
```

```
Query OK, 0 rows affected (1.80 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT emp_no,birth_date FROM employees WHERE first_name='Georgi' AND last_name='Facello' \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: First_And_Last_Name_IDX,First_And_Last_Name_DOB_IDX
          key: First_And_Last_Name_DOB_IDX
      key_len: 34
          ref: const,const
         rows: 2
        Extra: Using where; Using index
1 row in set (0.00 sec)
```

Now also we have Index on all the three columns first_name, last_name and birth_date. But there is change in order, the birth_date has been added at the end. So, while creating Index, you need to find out, how your query will look like, and accordingly add columns to the Index. In this case this query is using Covering Index, and hence will return the results faster.

Index Size - I

- Lower the index size, better the query performance.
  1. The more index (and data) records can fit into a single block of memory, the faster your queries will be.
- How to reduce Index size?
  1. By choosing best possible data type for columns in your table.
  2. Ask yourself questions like, Do I really need that BIGINT?, Can I replace this VARCHAR data type for particular field to INT or TINYINT?
  3. Did you know? An IP address can be reduced down to an UNSIGNED INT, so don't use char for IP.

Index Size - II

```
mysql> SHOW TABLE STATUS from empl200000db like 'employees' \G
```

```
*************************** 1. row ***************************
           Name: employees
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 200504
 Avg_row_length: 49
    Data_length: 9977856
Max_data_length: 0
   Index_length: 0
      Data_free: 30408704
 Auto_increment: NULL
    Create_time: 2015-02-24 23:43:09
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

Indicates length of Index in bytes. Since, we have not created any Index on employees, Index_length is 0 byte.

To know about SHOW TABLE STATUS in MySQL, refer the Bonus Slides at the end.

Index Size - III

```
mysql> ALTER TABLE employees ADD INDEX First_Name_IDX(first_name);
```

```
Query OK, 0 rows affected (1.44 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SHOW TABLE STATUS from empl200000db like 'employees' \G
```

```
*************************** 1. row ***************************
           Name: employees
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 200578
 Avg_row_length: 49
    Data_length: 9977856
Max_data_length: 0
   Index_length: 3686400
      Data_free: 27262976
 Auto_increment: NULL
    Create_time: 2015-02-24 23:52:31
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

Since we have created Index on the field first_name. Now this Index can be used by MySQL, whenever optimizer finds the Index useful for query execution.

Index Size - IV

```
mysql> ALTER TABLE employees ADD INDEX First_And_Last_Name_IDX(first_name,last_name);
```

```
Query OK, 0 rows affected (1.65 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SHOW TABLE STATUS from empl200000db like 'employees' \G
```

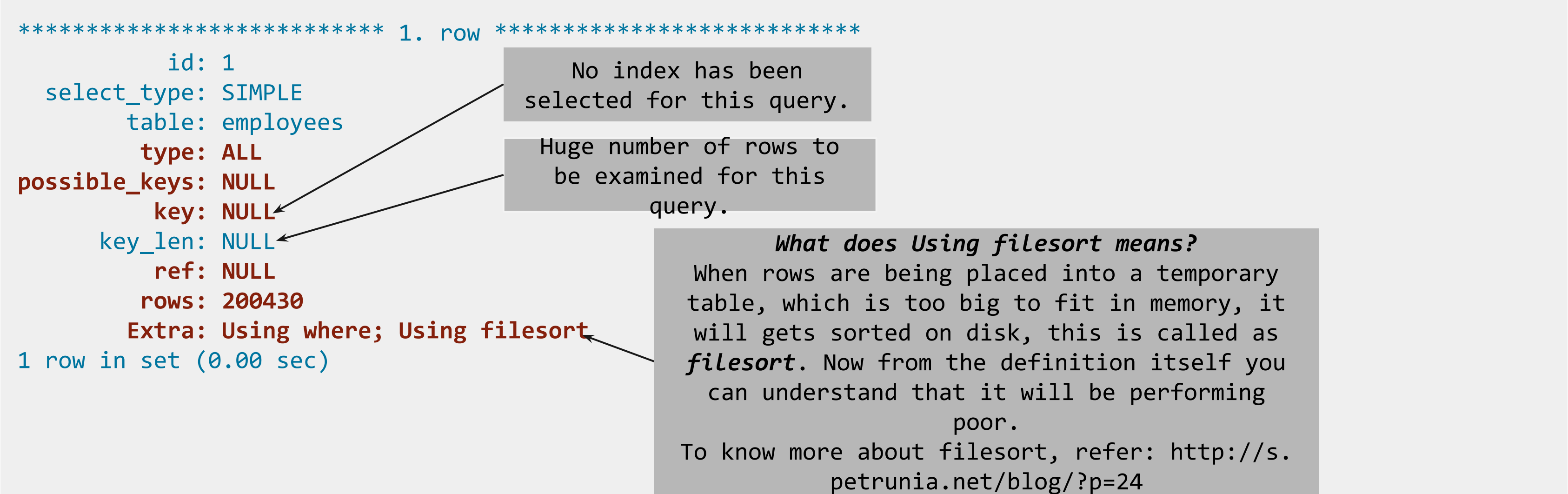
```
*************************** 1. row ***************************
           Name: employees
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 200357
 Avg_row_length: 49
    Data_length: 9977856
Max_data_length: 0
   Index_length: 9469952
      Data_free: 22020096
 Auto_increment: NULL
    Create_time: 2015-02-25 00:01:26
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

Now we have created Index also on first_name and last_name. Index length is growing rapidly. Always try to keep the Index size minimal for better performance.

Index can make your Sorting Faster - I

- In some cases, MySQL can use an index to satisfy an ORDER BY clause without doing any extra sorting.
- http://dev.mysql.com/doc/refman/5.0/en/order-by-optimization.html
- Examples:-

```
mysql> EXPLAIN SELECT * FROM employees where last_name = 'Facello' ORDER BY first_name \G
```



```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200430
        Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

No index has been selected for this query.

Huge number of rows to be examined for this query.

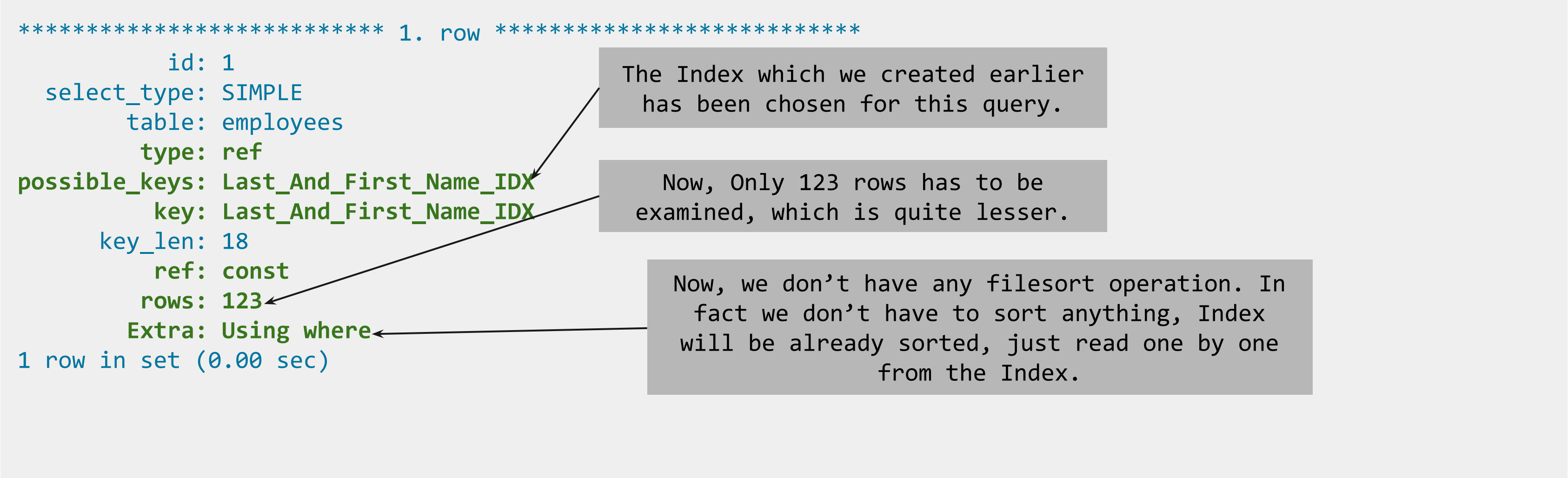
What does Using filesort means?
When rows are being placed into a temporary table, which is too big to fit in memory, it will gets sorted on disk, this is called as ***filesort***. Now from the definition itself you can understand that it will be performing poor.
To know more about filesort, refer: http://s.petrunia.net/blog/?p=24

Index can make your Sorting Faster - II

```
mysql> ALTER TABLE employees ADD INDEX Last_And_First_Name_IDX(last_name,first_name);
```

```
Query OK, 0 rows affected (1.88 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> EXPLAIN SELECT * FROM employees where last_name = 'Facello' ORDER BY first_name \G
```



```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
         type: ref
possible_keys: Last_And_First_Name_IDX
          key: Last_And_First_Name_IDX
      key_len: 18
          ref: const
         rows: 123
        Extra: Using where
1 row in set (0.00 sec)
```

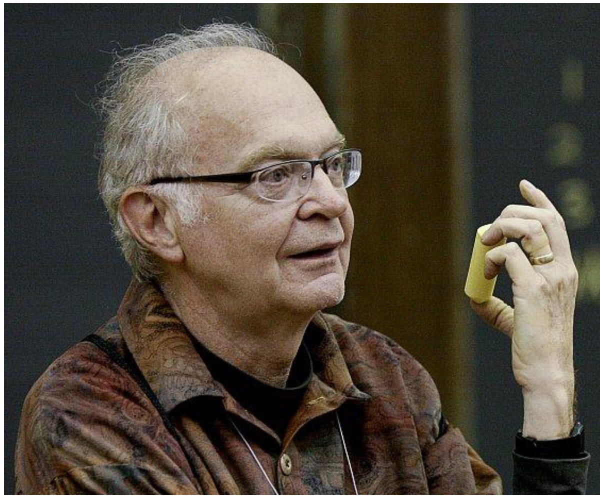
The Index which we created earlier has been chosen for this query.

Now, Only 123 rows has to be examined, which is quite lesser.

Now, we don’t have any filesort operation. In fact we don’t have to sort anything, Index will be already sorted, just read one by one from the Index.

Indexing Guidelines

- Create Indexes for set of your performance critical queries.
- It will be good, if all WHERE clause and JOIN clauses are using indexes for lookups.
- Try to extend Index, instead of creating new one.
- Always validate performance impact as you make the changes.
  “***An algorithm must be seen to be believed***”.
  - Donald Knuth



- Always choose best possible data type for the fields, that will be indexed.

Conclusion

By using Covering Index, we can dramatically improve query performance by reducing the I/O costs. Since the Index itself contains the required data field(s) and can return the data.
⚠ ***Warning***:

- It will cause extra memory overhead. ☹
- INSERT & UPDATE operation will slow down. ☹

Depending on your application use-case, whether your application is read intensive or write intensive. It is worth trying/analysing for most frequent read queries.

Resources

- Websites -

  http://planet.mysql.com/

  http://dev.mysql.com/doc/refman/5.6/en/

  http://www.percona.com/blog/

  https://mariadb.com/kb/en/mariadb/optimization-and-indexes/

  https://wiki.postgresql.org/wiki/Index-only_scans
- Books -

  High Performance MySQL by Baron Schwartz

  MySQL Database Design and Tuning by Robert Schneider

  SQL Antipatterns by Bill Karwin

  SQL Tuning by Dan Tow

Bonus Slides - I

What is MySQL Explain query?

- When you precede a SELECT statement with the keyword EXPLAIN, MySQL displays information from the optimizer about the statement execution plan.
- EXPLAIN helps us understand how and when MySQL will use indexes.
- EXPLAIN returns a table of data from which you identify potential improvements

To know more, please refer:

http://dev.mysql.com/doc/refman/5.6/en/using-explain.html

http://dev.mysql.com/doc/refman/5.6/en/explain-output.html

Bonus Slides - II

What is SHOW TABLE STATUS query?

- The SHOW TABLE STATUS statement displays many useful information about a table. For monitoring it can be very useful when dealing with particularly large tables.
- Information about Data_length and Index_length are the two, which is quite useful

To know more, please refer:

http://dev.mysql.com/doc/refman/5.6/en/show-table-status.html

Thank You !