Optimized cluster index generation for a table through observation of table operations on database.

CMPE 272 Group Project - **Team 13**

Aman Ojha
Software Engineering Department
San Jose State University
San Jose, USA
aman.ojha@sjsu.edu

Rutvik Pensionwar
Software Engineering Department
San Jose State University
San Jose, USA
rutvik.pensionwar@sjsu.edu

Arijit Mandal
Software Engineering Department
San Jose State University
San Jose, USA
arijit.mandal@sjsu.edu

Vishweshkumar Patel
Software Engineering Department
San Jose State University
San Jose, USA
vishweshkumar.patel@sjsu.edu

***Abstract*—A cluster index is a structure that sorts and stores the entries in the table and expedite the process of data retrieval in table and views. Indexing is now one of the vital components in the data retrieval process lifecycle. Companies nowadays prioritize indexing upon other tasks to achieve faster access to the data and attenuate the time required for query execution. However, a data store of a global company is associated with myriad tables and views where generating the most optimized cluster index becomes challenging. So, we as a team have developed an application that logs end user's queries over time and eventually parse it on the basis of an algorithm (refer 'Algorithm' section of the report). This will assist the data manager to set the most effective column as a cluster index and accelerate the data acquisition process.**

***Index Terms*—Parsing, MySQL, B-tree, Hash, data logging**

I. INTRODUCTION

Indexes are used to expedite the query execution time without which the pointer would traverse from the first row till the last entry of the table/view to notice the relevant row. Larger the database (table), more is the cost of execution. However, if the index is already set, irrelevant entries are skipped, thus advancing the search.

Types of indexes

1. B-Tree
    a. multi-purpose, most-popular, index data structure
    b. leaf rows are sorted
    c. supports single-row lookups
    d. supports range lookups
    e. supports scanning in order (sorted access)

2. Hash
    a. relatively special-purpose
    b. supports single-lookups
    c. doesn't support ordered access, bulk access, scanning

3. Log-Structured Merge
    a. general-purpose family of algorithms & data structures
    b. supports single-row, bulk, and sorted access
    c. avoids few problems with B-tree indexes (adds complexity)

4. Other (full-text, spatial, skiplist)

II. HOW MYSQL MAKES USE OF INDEX

1. finds rows related to columns present in where clause
2. upon multiple indexes, MySQL uses one which acquires the smallest number of rows
3. on multiple columned index, MySQL have indexed search capabilities on all its permutations.
4. while using joins between two tables, indexes can be used more effectively if the columns on which join condition is enforced are of same type.
5. if a query is written in such a way that it mentions only the column that is indexed, entries can be retrieved at much better speed.

Note: For some of the cases, indexes aren't that critical; when query needs to access majority of rows, sequential traversal is efficient than search based on index.

To read data in bulk, we can use clustered indexes. A clustered index means that the tree structure terminates in the table rows itself. The table itself is stored in sorted order. InnoDB supports this implicitly by clustering on the primary key.

Primary Key Optimization

The primary key is the unique column of your table with highest usage frequency all over the DB.MySQL maintains an associated index here. Optimization is high-up because the primary key is never NULL. Moreover, InnoDB allows data to be organized for faster lookups, groups, and sorts based on such primary key.

Foreign Key Optimization

Views generally are composed of several columns, however frequency of operations are concentrated over a few. Rest are never touched. An efficient way to decrease the execution time is to have those isolated columns into a separate table and associate them with parent table using a key. This will perform less I/O and end up releasing cache memory.

Column Indexes
A B-tree data structure is responsible for the faster lookups while using indexes. It quickly tracks a value or set/range of values which corresponds to various arithmetic operations in WHERE clause.

III. HOW TO GET MORE FROM INDEXES

1. Make it sargable
    a. make sure that the query's expressions can actually be used as keys in index.
    b. manage columns in WHERE clause

2. Left Prefix Rule
    a. note that the multi-column indexes are sorted by first-column first and then the secondary one.
    b. index prefixes used must start from left
    c. the index prefix seems irrelevant at the first non-equality differentiation
    d. indexes must be aligned in such a way that the equality comparison columns comes first, followed by other comparisons
    e. prefer 'IN' instead of range, if range is relatively small

3. Index only queries
    a. form a covering index for most frequent queries

4. Use clustered indexes
    a. you have one and only one clustered index in InnoDB
    b. avoid using auto-incremented values if you have a primary key associated with the table

5. Column Order
    a. you can optimize ORDER BY, GROUP BY, and WHERE clause by including correct column order
    b. place most selective column in front, if preference for WHERE clause is absent
    c. try to serve several queries with common set of indexes

6. Don't Over-index
    a. more indexes, more write cost
    b. take care there aren't duplicate indexes
    c. drop least recently used or unused index

Normally, statement 'ALTER TABLE' is used to create to indexes on a specific column/s, but InnoDB gives us the priviledge to add indexes while creating a table. Thus, a single or multiple-column indexes can be defined while executing the above DDL command.

eg: Single-column index:
```
ALTER TABLE `user_with_index`.`address`
ADD INDEX `index_addId_1` (`addressId` ASC);
```

eg: Multiple-column index:
```
ALTER TABLE `user_with_index`.`user`
ADD INDEX `index_1` (`user_id` ASC),
ADD INDEX `index_2` (`country_code` ASC);
```

IV. WHAT'S THE PROBLEM WE ARE FACING

1.Performance of SQL database while employing them to perform real time analysis on regular basis

2.´Delay in response time while performing complex queries on large amount of transactional data stored in SQL databases – join, order by, group by, where clause

3.´Factors to decide clustering index for SQL table –

-Is table indexed based on usage?

-How often will this query or set of query, be executed?

-How much time or I/O will be saved?

-Repetitive usage of set of queries in particular timeline

V. TARGETED USER GROUP

1. Insurance firms - State farms

2. Investment firms - Morgan Stanley, Goldman Sachs, etc

3. Pharmaceutical firms - Novartis, Johnson & Johnson, etc

4. Product based firms – IBM, Apple, Google, Amazon, etc

5. Service based firms – IBM, PWC, Deloitte, etc

6. Basically, all firms that depends on transactional data analysis on repetitive basis

VI. OUR SOLUTION

1. Observe queries being performed on tables

2. Differentiate complex queries and log them

3. Parse logged queries and find weights of columns and based on that find best column for a table to be considered for cluster index

4.Store relevant results for future analysis

5. Upon need to perform same analysis, reindex tables based on previously found results.

VII. ALGORITHM

As mentioned earlier, queries fired from the end user are repeatedly logged and parsed in order to analyze the optimized clustered index. Initially, all the table names and their respective columns which appear after WHERE clause are registered. The ones appearing after ORDER BY and GROUP BY are also noted down. Since sorting/grouping is very resource intensive operation, it takes considerable amount of CPU times as compared to others. This is because a typical sort operation must read the entire input before delivering the first output. Since sort operations cannot be applied in sequential manner, it can be an overhead for large databases.

For this reason, we double the weight for all those columns mentioned after an ORDER/GROUP BY clause. The final weights assigned for each clause are mentioned below:

Query	Weight
WHERE	1
ORDER BY	2
GROUP BY	2

Once the complete query is parsed, the columns of respective tables are weighted against the algorithm specified. If multiple tables are used, the output of the parsing algorithm lists all its individual columns and their weights. A column with highest weight amongst its table, will be the index.

VIII. PARSING LOGIC

We observe the various queries which are being performed on the tables. After careful analysis, we log the complex queries into a log file which is used for performing clustering operation. We parse the log and give weight to the operation performed based on the complexity and resources required by the operation. In the log, each row is read and key variables like table name, column name, operation performed etc are stored. Then, we analyse which column is used the most and also the column on which various complex queries are performed. At the end of parsing, we find the best suitable column which is to be considered for clustering.

After deciding on the columns, we perform indexing operation on the MySQL to further enhance the performance of data retrieval.

IX. SIMULATION RESULTS

We have tested the application under multiple circumstances. Various permutations include applying clustering index on multiple columns of a table and compare it with table without any index. The optimization results were impressive. Analysis charts show a performance improvement in query execution as follows:

TABLE 1: DETAILED QUERY WITH DEFINITION

Query	Description
Query 1	Show user's country details select user.first_name, user.last_name, user.company, user.country_code, country.country from user, country where **user.country_code = country.country_code** order by user.user_id
Query 2	Show user's company stock details select user.first_name, user.last_name, user.company, stock.stock_sector, stock.stock_symbol, stock.stock_market_cap from user, stock where **user.company = stock.company** and **stock.stock_sector in ('Technology', 'Finance')**
Query 3	Show male:female count(countrywise) select country.country, user.gender, count(*) from user, country where **user.country_code = country.country_code** group by **country.country, user.gender** order by **user.country_code**
Query 4	Show user's company details select user.first_name, user.last_name, user.company, company.street_number, company.street_name, company.city, company.state, company.country from user, company where **user.company = company.company** order by **user.user_id**
Query 5	Show user address details select user.user_id, user.first_name, user.last_name, user.addressId, address.street_name, address.city, address.state from user, address where **user.addressId = address.addressId** order by **user.user_id**

Comparing query execution time with & w/o index:

TABLE 2: ACTUAL EXECUTION TIME WITH/WITHOUT INDEXING

Query	Execution Time		
	Without index	*With index*	*% reduction in time*
Query 1	0.12254325	0.04865050	60.29 %
Query 2	0.84448975	0.84462825	-0.01 %
Query 3	0.12020500	0.05333575	55.62 %
Query 4	4.64891075	0.05071725	98.90 %
Query 5	7.81886425	0.04992700	99.36 %

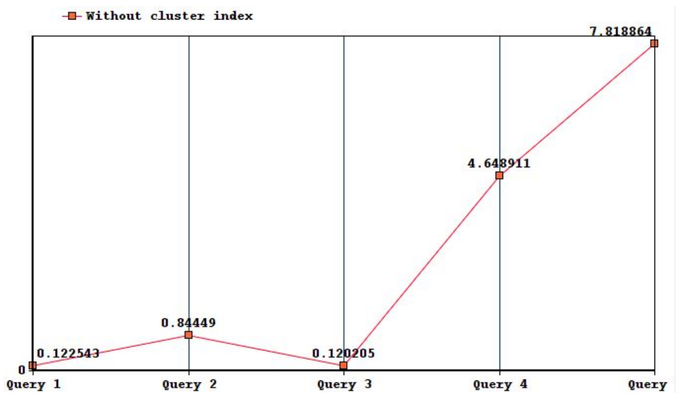

Fig: The execution time required per query (without index)

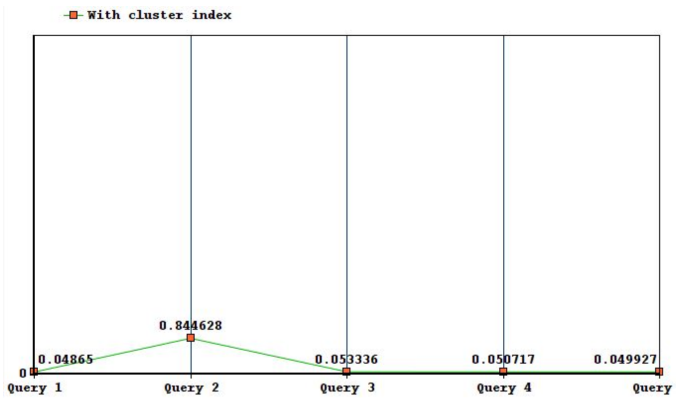

Fig: The execution time required per query (with index)

The figure below clearly depicts the difference in execution time of queries (all 5) with and without index.

X. RESULT PRESENTATION

We have used chartjs to show our analysis/results. Chart.js is another JavaScript library that helps us to draw different types of charts by using the HTML5 canvas element. The final results with and without indexing are as shown below:

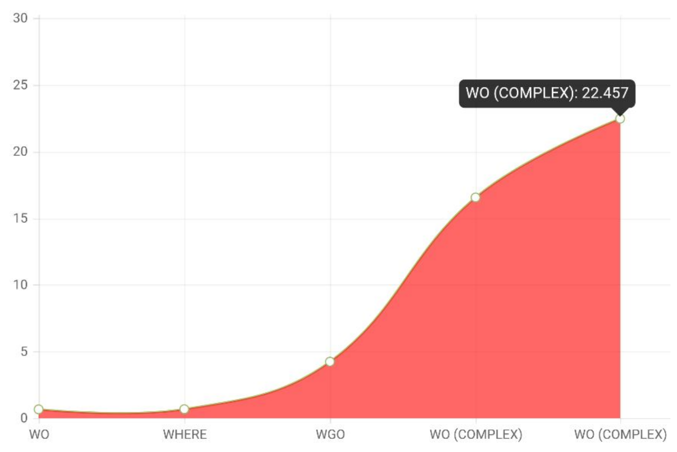

Fig: Query execution time w/o index (Y axis scale 0-30ms)

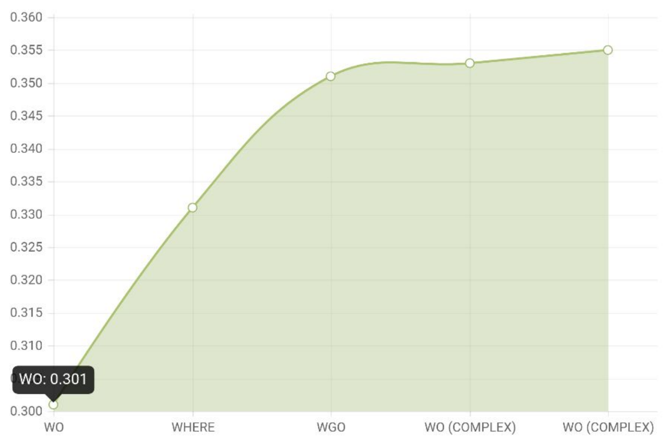

Fig: Query execution time with index (Y axis scale 0-0.360ms)

XI. ACKNOWLEDGEMENT

Using the algorithm described above, we were able to implement the problem statement successfully. The column name (for individual table) was derived in such a way that the query execution time was minimized when it acts as an index. The average reduction in execution time (all 5 queries) is about 62.8 % which seems pretty appreciable.

XII. REFERENCES

[1] Optimizing MySQL Queries With Indexes Baron Schwartz, VividCortex March 17th, 2015