International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

MySQL Select Query Optimization Using Self-Join

Most optimized query execution for fetching particular results in a large database with millions of records

Chitransh Madavi[1], Chetan Patel[2], Vedant Jain[3], Prof. Vandana Kate[4]

[1,2,3,4]Department of CSIT, Acropolis Institute of Technology and Research, Indore, India

DOI: https://doi.org/10.55248/gengpi.234.4.36035

ABSTRACT—

The challenge of query execution in MySQL involves the efficient and accurate processing of SQL queries to retrieve, manipulate, and store data in a database. It requires optimizing the query execution plan, identifying and resolving performance issues, and ensuring data integrity and security. The challenge lies in balancing the trade-offs between query complexity, database size, system resources, and user requirements. Choosing the right approach depends on the specific characteristics of the data and workload. Therefore, in this paper, we have applied various techniques to get optimized results.

Keywords—Database, Query Optimization, DBMS, MySQL, Self-Join, indexing

I. Query Optimization

Query optimization refers to the process of improving the performance and efficiency of a database system when executing a query. It involves analyzing and restructuring the query to reduce the amount of time and resources required to retrieve, manipulate, and store data. The goal of query optimization is to produce the same result set as the original query, but with faster execution time and minimal resource utilization. This is achieved by selecting the optimal query execution plan, which involves considering factors such as table access methods, join algorithms, and index usage. In short, query optimization is about making queries run faster and consume fewer resources without changing the results they produce.

II. Problem statement

To generate an optimized MySQL query fulfilling the below requirements:

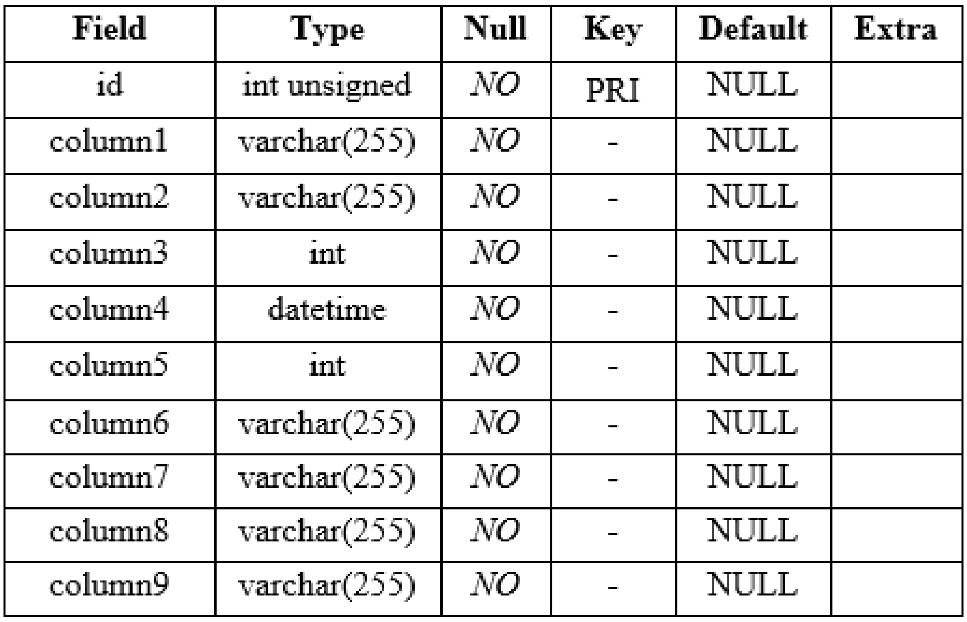
A. Database and Schema structure

The database should be configured standardly with InnoDB engine, while table structure can contain at least ten columns with different data types and around 5-6 million of records. The initial schema structure is depicted in "Tab.1".

B. Result requirements

- At least two 'WHERE' conditions with the 'AND' operator should be satisfied
- The fetched result should be sorted with any column.
- The required number of columns to be selected can be less than six.
- Selected results should be feasible of pagination i.e., limit along with a very variable offset should be applied.

TABLE.1 : INITIAL SCHEMA STRUCTURE OF MYSQL TABLE



Field	Type	Null	Key	Default	Extra
id	int unsigned	*NO*	PRI	NULL	
column1	varchar(255)	*NO*	-	NULL	
column2	varchar(255)	*NO*	-	NULL	
column3	int	*NO*	-	NULL	
column4	datetime	*NO*	-	NULL	
column5	int	*NO*	-	NULL	
column6	varchar(255)	*NO*	-	NULL	
column7	varchar(255)	*NO*	-	NULL	
column8	varchar(255)	*NO*	-	NULL	
column9	varchar(255)	*NO*	-	NULL	

III. Literature review

Query Optimization In MySQL Database Using Index[10]

The findings of this study revealed that the quality of the database design has a substantial influence on the quality of the information system. Even after a system has been in use for some time, a programmer could forget to optimize a MySQL query, even if optimization is crucial owing to the enormous amount of data. MySQL's index function for tables helps to speed up database searches. Alternatively, the index, built at random and in large quantities, would slow down database access if it did not serve the user's needs and interests. So, it is crucial to go over the instructions and conduct a rigorous examination before creating a database index.

A Study on Join Operations in MongoDB Preserving Collections Data Models[1]

According to this report, there is currently an enormous amount of data that needs to be processed and stored online. High volume, high velocity, and high variety characterize this data. As a result, NoSQL data storage alternatives have begun to be taken into account by software developers. Operations that are straightforward in traditional Relational Database Management Systems (DBMSs) might, however, become rather challenging with NoSQL DBMSs. Several NoSQL databases do not explicitly support the join operation, which connects two or more DB structures. As a result, changing the data model or taking a number of steps is required to respond to particular data queries. They provided a technique for carrying out join operations in MongoDB at the application layer, allowing us to specifically maintain data models. They assessed the performance of the suggested method and compared the extra overhead to SQL-like databases.

Join Query Processing in Data Quality Management [12]

Here, the observation was that the Management of data quality is the key issue facing information systems. Joins on large-scale data are a fundamental component of data quality management, and they are crucial to 'document clustering'. A programming approach called MapReduce is frequently used to process enormous amounts of data. The framework allows for the implementation of numerous tasks, including machine learning and search engine data processing. Nevertheless, current implementations do not allow join operations in an efficient manner.

Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases [3]

In this study, various strategies for enhancing the database structure and queries were applied to two popular open-source database management systems—MySQL as a relational DBMS and document-based MySQL as a non-relational DBMS. The primary objective of the study was to examine and contrast how the suggested optimization approaches affect each specific DBMS when handling CRUD (CREATE, READ, UPDATE, DELETE) queries. The analysis and performance assessment of CRUD activities for different data quantities were carried out using a case study testing architecture built on Java. The results show how much the recommended optimization strategies enhanced application performance for both databases; on the basis of these, a complete analysis and a number of recommendations are made to support a particular course of action.

IV. Conventional practices

Optimizing a SELECT query involves analyzing and improving the query execution plan to retrieve the required data with minimal time and resource utilization. Some of the common conventional practices to optimize a SELECT query are as follows:

Use Indexes

Indexes help to speed up the data retrieval process by allowing the database to quickly locate the required data. Ensure that the columns used in the WHERE and JOIN clauses are indexed.

Use LIMIT clause

The LIMIT clause helps to reduce the number of rows returned by the query, thus reducing the processing time and resource consumption. It is especially useful when dealing with large tables.

*Avoid using SELECT **

Only select the required columns instead of selecting all columns in the table. This reduces the amount of data retrieved and processed, leading to faster execution time.

Use subqueries

Only select the required columns instead of selecting all columns in the table. This reduces the amount of data retrieved and processed, leading to faster execution time.

Use caching

Caching the query results can help to avoid repeated execution of the query, thus reducing the processing time and resource utilization

Optimize table structure

Normalizing the table structure, partitioning large tables, and using appropriate data types can help to improve the query performance.

Analyze query performance

Use tools such as 'EXPLAIN' to analyze the query execution plan and identify areas that need optimization.

V. Optimization using conventional methods

So the example of the initial query for the above requirements can be framed as –

*"SELECT * FROM table_name WHERE `column1`="value1" AND `column2`="value2" ORDER BY column9"* **(1)**

The above query executed well but the problem that arose is the slower fetch time as the number of records is very high fetching the results is very slow with any optimization technique. So, we will apply optimizations one by one and analyse:

Not using SELECT *

Since we do not require every column to be fetched, we can specify the exact columns to be fetched. Suppose we required only – columns3, columns4, column5 and columns6. So, the optimized query can be:

*SELECT column3,column4,column5,column6 FROM table_name WHERE `column1` = "value1" AND `column2` = "value2" ORDER BY column6"***(2)**

The above query resulted in less fetch time and execution time because of the decrease in the required data.

Still, the optimization is not sufficient so we need to move further.

Use LIMIT clause

Since we required the result for pagination, fetching all the results at one time is a waste of resources. Suppose we need the 400th page with 25results each, the optimized query with limit and offset can be:

"*SELECT column3,column4,column5,column6 FROM table_name WHERE `column1` = "value1" AND `column2` = "value2" ORDER BY column6 LIMIT 25 OFFSET 9975*" **(3)**

TABLE.2 : EXPLAINED VERSION OF QUERY 1, QUERY 2, AND QUERY 3

Column Name	Query Value
id	1
select_type	SIMPLE
table	table_name
Partitions	-
type	ALL
possible_keys	-
key	-
key_len	-
ref	-
rows	2552027
filtered	1.00
Extra	Using where; Using filesort

Use Indexes

Since we are applying the 'WHERE' condition on the 'column' and 'column2''ORDER BY' sort on 'column9'. We can create an index on the above-mentioned columns for faster search. The query will remain the same.

TABLE.3 : EXPLAINED VERSION OF QUERY 3 WITH INDEXING

Column Name	Explain Value
id	1
select_type	SIMPLE
table	table_name
Partitions	-
type	ref
possible_keys	column1_index,column2_index
key	column1_index
key_len	767
ref	const
rows	1054504
filtered	41.32

TABLE.4 : SCHEMA STRUCTURE OF TABLE AFTER INDEXING

Field	Type	Null	Key	Default	Extra
id	int unsigned	*NO*	PRI	NULL	
column1	varchar(255)	*NO*	MUL	NULL	
column2	varchar(255)	*NO*	MUL	NULL	
column3	int	*NO*	-	NULL	
column4	datetime	*NO*	-	NULL	
column5	Int	*NO*	-	NULL	
column6	varchar(255)	*NO*	-	NULL	
column7	varchar(255)	*NO*	-	NULL	
column8	varchar(255)	*NO*	-	NULL	
column9	varchar(255)	*NO*	MUL	NULL	

UsingSubquery

We can get the same result using subquery also, the query formed will be:

"SELECT column3,column4,column5,column6 FROM table_name WHERE id IN (SELECT id FROM table_name WHERE `column1` = "value1" AND `column2` = "value2") LIMIT 25 OFFSET 9975" ***(4)***

TABLE.5 : EXPLAINED VERSION OF QUERY 4

Column Name	Sub Query Value	Query Value
id	1	1
select_type	SIMPLE	SIMPLE
table	table_name	table_name
Partitions	NULL	NULL
type	ref	eq_ref
possible_keys	PRIMARY,column1_index ,column2_index	PRIMARY
key	column1_index	PRIMARY
key_len	767	4
ref	const	table_name.id
rows	1054504	1
filtered	41.32	100.00
Extra	Using where	NULL

VI. Optimization Using Self-Join

Conventionally a join operation is considered as expensive in terms of using resources but in this case even after trying various approaches the execution and fetching time of the query is not sufficient.

But now using self-join along with the above technique instead of a subquery, both the execution and fetching time reduces significantly. The reason for the decrease in execution time is due to mapping only the selected 'id' in the final selection of the result without having to iterate throughout the unwanted results. The query can be framed as:

"*SELECT A.column3,A.column4,A.column5,A.column6 FROM (SELECT id FROM table_name WHERE `column1` = "value1" AND `column2` = "value2" LIMIT 25 OFFSET 9975) B JOIN table_name A ON A.id = B.id"* *(5)*

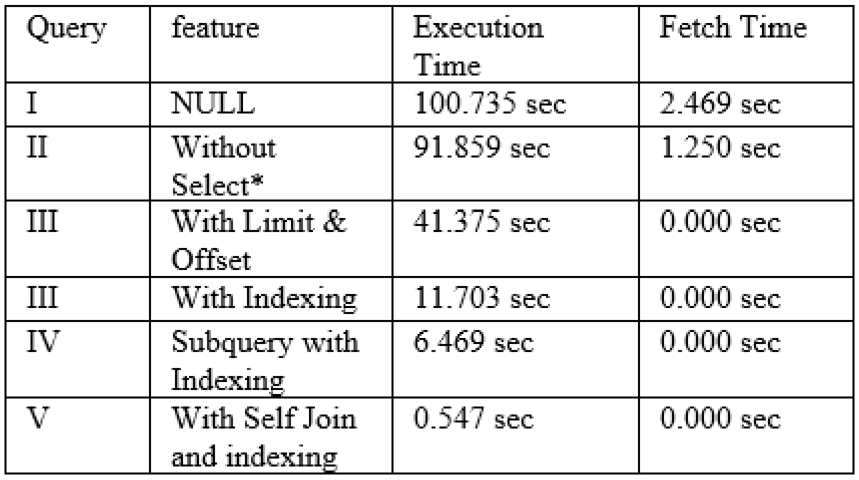
TABLE.6 : EXPLAINED VERSION OF QUERY 5 (MOST OPTIMAL)

Column Name	Value	Value	Value
id	1	1	2
select_type	PRIMARY	SIMPLE	DERIVED
table	<derived2>	A	table_name
Partitions	NULL	NULL	NULL
type	ALL	eq_ref	ref
possible_keys	NULL	PRIMARY	column1_index, column2_index
key	NULL	PRIMARY	column1_index
key_len	NULL	4	767
ref	NULL	B.id	const
rows	10000	1	1054504
filtered	100.00	100.00	41.32
Extra	NULL	NULL	Using where

VII. RESULT ANALYSIS

After successfully executing the all above gradually optimized MySQL queries, the following result is listed in the table given below:

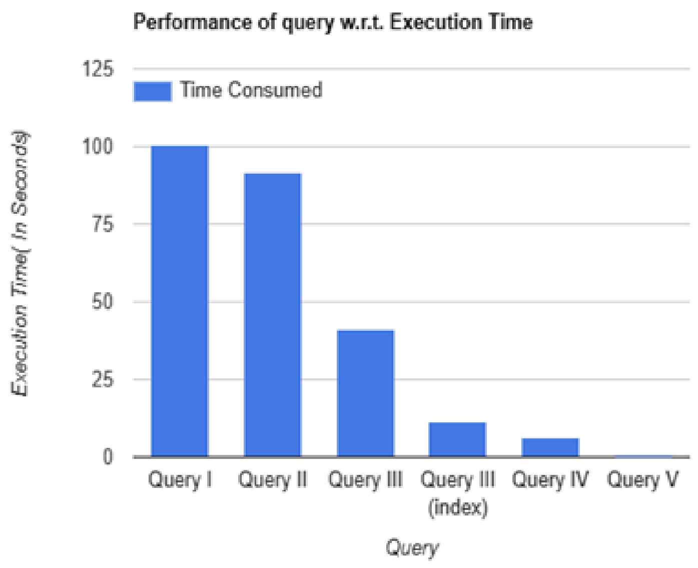
TABLE.7 : QUERY PERFORMANCE FOR VARIOUS FEATURE



Query	feature	Execution Time	Fetch Time
I	NULL	100.735 sec	2.469 sec
II	Without Select*	91.859 sec	1.250 sec
III	With Limit & Offset	41.375 sec	0.000 sec
III	With Indexing	11.703 sec	0.000 sec
IV	Subquery with Indexing	6.469 sec	0.000 sec
V	With Self Join and indexing	0.547 sec	0.000 sec

The results can be depicted graphically as depicted below:

Fig.1: Relationship of queries with Fetch time

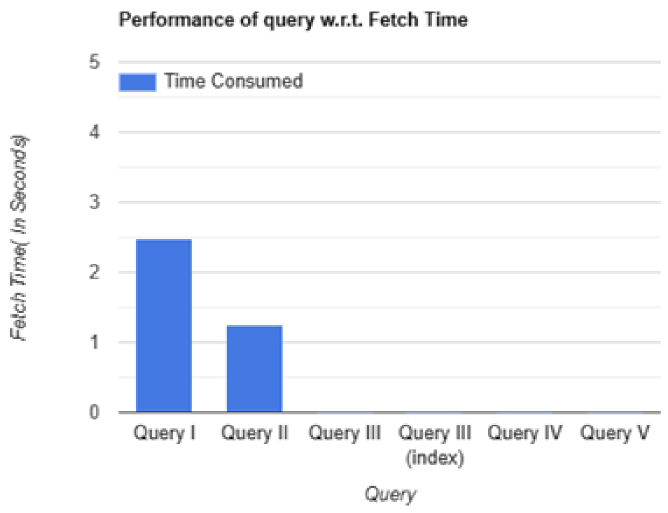


It can be clearly observed in Fig.1 that fetch time reduced significantly since Query III, after applying the LIMIT operation. It happened because the LIMIT operation restricts the number of rows returned by the query. By returning a smaller subset of rows, the database engine can avoid the need to process and return the entire result set, which can be a time-consuming and resource-intensive operation.

When a query without the LIMIT operation is executed, the database engine needs to retrieve and process all matching rows in the table(s) before returning the result set. This can take a significant amount of time and consume a large number of resources, especially when dealing with large tables or complex queries. However, when the same query is executed with the LIMIT operation, the database engine only needs to retrieve and process a limited number of rows specified by the LIMIT value. This results in a faster fetch time and reduced resource consumption, as the database engine can stop processing and returning rows once the LIMIT value is reached.

From Fig.2, it can be observed that the execution time is reducing gradually operation by operation but while executing Query V i.e. with Self-join, Execution time reduces significantly executing the query in less than a second.

Fig2: relationship of queries with fetch time



In conclusion, optimizing the data-reading process can improve the performance of retrieving rows from a database table. However, other factors such as the size of the database, indexing, network latency, and server load can also affect the performance of retrieving rows. Therefore, it is important to consider these factors when optimizing the data-reading process and writing efficient queries. In the case of LIMIT queries, limiting the number of rows returned can significantly reduce the amount of data that needs to be processed, which can help to improve performance.

VIII. CONCLUSION

In conclusion, self-join is a powerful technique that can be used to optimize MySQL queries that involve the same table multiple times. By using self-join, you can avoid the need to create temporary tables or use subqueries, which can lead to slower query performance and higher resource utilization.

When using self-join, it is important to consider the query design and optimize the query execution plan to minimize the number of rows and columns processed. This can be achieved by properly indexing the tables, selecting the appropriate join type, and limiting the number of rows returned using the WHERE clause.

Self-join can also be used in conjunction with other optimization techniques such as query caching, query tuning, and partitioning to further improve query performance and scalability.

Overall, self-join is a valuable tool in the MySQL optimization toolbox that can help improve query performance and reduce resource utilization, but it should be used judiciously and in conjunction with other optimization techniques to achieve the best results.

Acknowledgement

We would like to express our gratitude to our mentor Prof. Vandana Kate for her unparalleled mentorship. Major thanks to the Department of CSIT and HOD Dr. Shilpa Bhalerao for their guidance, advice throughout the research process and for always supporting our ideas and opinions.

We would also like to thank the reviewers for their constructive feedback and suggestions, which helped to improve the quality of this paper.

References

[1]. Celesti, M. Fazio, and M. Villari, “A Study on Join Operations in MongoDB Preserving Collections Data Models for Future Internet Applications,” Future Internet, vol. 11, no. 4, p. 83, Mar. 2019, doi: 10.3390/fi11040083.

[2]. Akerkar, R. (Ed.). (2013). Big Data Computing (1st ed.). Chapman and Hall/CRC. https://doi.org/10.1201/b16014.

[3]. A. Győrödi, D. V. Dumşe-Burescu, R. Ş. Győrödi, D. R. Zmaranda, L. Bandici, and D. E. Popescu, “Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases,” Applied Sciences, vol. 11, no. 15, p. 6794, Jul. 2021, doi: https://doi.org/10.3390/app11156794

[4]. J. D. Zawodny and D. J. Balling, High Performance MySQL. “O’Reilly Media, Inc.,” 2004..

[5]. J. F. García, "MQOP-A Tiny Reference to the Multiple-Query Optimization Problem," Revista de.

[6]. K. O'Gorman, A. El Abbadi, and D. Agrawal, "Multiple query optimization in middleware using query teamwork," Software: Practice and Experience, vol. 35, no. 4, pp. 361–391, 2005, doi: https://doi.org/10.1002/spe.640.

[7]. R. Sahal, M. H. Khafagy, and F. A. Omara, "Comparative Study of Multi-query Optimization Techniques using Shared Predicate-based for Big Data," International Journal of Grid and Distributed Computing, vol. 9, no. 5, pp. 229–240, May 2016, doi: https://doi.org/10.14257/ijgdc.2016.9.5.20.

[8]. R. Sahal, M. Nihad, M. H. Khafagy, and F. A. Omara, "iHOME: Index-Based JOIN Query Optimization for Limited Big Data Storage," Journal of Grid Computing, vol. 16, no. 2, pp. 345–380, Mar. 2018, doi: https://doi.org/10.1007/s10723-018-9431-9.

[9]. R. Sahal, M. H. Khafagy, and F. A. Omara, "Exploiting coarse-grained reused-based opportunities in Big Data multi-query optimization," Journal of Computational Science, vol. 26, pp. 432–452, May 2018, doi: https://doi.org/10.1016/j.jocs.2017.05.023.

[10]. S. Maesaroh, H. Gunawan, A. Lestari, M. S. . A. Tsaurie, and M. Fauji, "Query Optimization In MySQL Database Using Index", IJCITSM, vol. 2, no. 2, pp. 104–110, Mar. 2022.

[11]. X. Zhang, L. Chen, and M. Wang, "Efficient multi-way theta-join processing using MapReduce," Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1184–1195, Jul. 2012, doi: https://doi.org/10.14778/2350229.2350238.

[12]. Yue, M., Gao, H., Shi, S., Wang, H.: Join query processing in data quality management. In: Database Systems for Advanced Applications, pp. 329–342. Springer International Publishing, Cham (2016)