Optimize Join Implementation for All Version of MySQL

Group 5

Yi-Hsien Chen
Department of Electrical Engineering
National Taiwan University Taipei,
Taiwan f08921a05@ntu.edu.tw

Yin-Tzu Huang
Department of Electrical Engineering
National Taiwan University Taipei,
Taiwan d06921008@ntu.edu.tw

Yih-Lin Liu
Department of Biomechatronics
Engineering National Taiwan
University Taipei, Taiwan
r11631028@ntu.edu.tw

Hui-Wen Chuang
Graduate Institute of Medical
Genomics and Proteomics, National
Taiwan University Taipei, Taiwan
r08424026l@ntu.edu.tw

Pin-Xuan Chen
Graduate Institute of Medical
Genomics and Proteomics, National
Taiwan University Taipei, Taiwan
r10455001@ntu.edu.tw

ABSTRACT

Our research focuses on enhancing the performance of join queries by optimizing the join process. The traditional approach involves retrieving all attributes from the entire table and then filtering out unnecessary information, leading to wasted storage space. To address this, we propose preselecting the required attributes in advance, resulting in significant time and space savings. Additionally, we aim to improve the efficiency of multi-table joins, which are currently time-consuming due to nested joins. To achieve this, we introduce a red-black tree as a data structure, built based on match conditions during joins. This allows us to efficiently search for and retrieve matching data during subsequent join operations. Furthermore, we optimize memory usage by storing references to the original data instead of duplicating it for each row. This approach reduces redundancy and improves memory utilization, particularly for large datasets. By eliminating unnecessary data duplication and implementing these optimizations, we enhance the overall efficiency of the join process.

KEYWORDS

Join optiminzation, Condition matching, Red black tree, MySQL

1 INTRODUCTION

In the ever-evolving landscape of data analytics, processing queries on large and complex datasets poses significant challenges. To address these challenges, advanced query-processing technologies have emerged, focusing on efficient handling of substantial queries. While substantial progress has been made in areas such as sophisticated data layouts, scalable systems, and JIT code generation, the optimization of query processing modules has not received commensurate attention. This discrepancy is further emphasized by the description of existing systems' limitations in handling large query scenarios [1]. Consequently, finding an optimal plan for queries involving a large number of relations remains a formidable task due to the exponential growth of the search space with relation count [2].

In light of these considerations, our research aims to enhance the performance of join queries. The conventional approach involves retrieving all attributes from the entire table and subsequently filtering out unnecessary information using conditions like "select" and "where." However, this approach has several drawbacks, including the selection and processing of redundant attributes, leading to inefficient utilization of storage space.

1.1 Motivation

Our goal in this research is to improve the performance of join query. The original join method involves fetching all attributes from the entire table and then filtering out unwanted information using conditions (e.g. select, where). However, this approach can lead to several issues. For example, many attributes are actually unnecessary during this process, yet they are still selected and processed, resulting in wasted storage space. Therefore, if we can preselect the attributes that will be truly needed in advance, it can save a significant amount of time and space. In addition, the conventional approach to joining tables involves using nested joins, which generates all possible combinations and then filters the data based on conditions. However, this method is highly time-consuming. As a result, in this study we want to find an optimized method to make a multi-table JOIN process efficient.

1.2 Contributions

Our goal is to optimize the join process . Specifically, we aim to:

1) Select and store attributes that are used in the following steps. Instead of bringing the entire table's attributes for the join process, we can optimize by selecting and storing only the necessary attributes. This reduces the amount of data being processed and saves storage space.

2) Our approach involves building a red-black tree. The conditions for building the red-black tree is based on the match conditions during the join process. Therefore, during future joins, we can directly search on the red-black tree and retrieve the matching data for the join process.
3) Store the address of the original data instead of copying it for each row: Rather than duplicating the data for each row during the join process, we can optimize by storing the address or reference to the original data. This approach reduces redundancy and saves memory resources, especially when dealing with large datasets. By referencing the original data, we can access and process it directly, avoiding unnecessary data duplication and improving overall efficiency.

2 RELATED WORKS

2.1 Hash join

Hash join is implemented by first creating a hash table from one of the input tables, using the join key as the hash function. Then, the other table is scanned, and each row is matched with the corresponding entries in the hash table. This allows for efficient retrieval of matching rows, resulting in faster join operations. Hash join is widely used in database systems to optimize join performance for large datasets.

2.2 Merge join

Merge join is implemented by sorting both input tables based on the join key and then merging them. It compares the values of the join keys in sorted order and combines matching rows. Merge join requires the input tables to be pre-sorted, but it avoids the need for additional data structures like hash tables. It is commonly used for join operations in database systems when the input tables are already sorted or when the available memory is limited. Merge join provides efficient join performance for large datasets.

2.3 Broadcast join

Broadcast join is implemented by broadcasting a small table to all nodes in a distributed system and joining it with a larger table. The small table is replicated across all nodes, eliminating the need for data shuffling. Each node then performs a local join between the broadcasted table and its portion of the larger table. Broadcast join is suitable when the small table can fit into memory, and the larger table is partitioned across multiple nodes. It reduces network overhead and improves join performance by minimizing data movement. Broadcast join is commonly used in distributed systems for efficient join operations.

2.4 Compare with other SQL language

In the past, MySQL basically supported only nested loop joins [4]. This was because MySQL was based on the design philosophy of “I do not support complex algorithms as much as possible”. MySQL was originally used in embedded systems before it was used in web applications. It’s necessary to run the database in a disk or memory with a very small capacity of the embedded device, and it has been designed with the policy of dropping complicated algorithms as much as possible. But the new version of MySQL adds a new algorithm, Hash Join and sort-merge join.

In Postgres, the relationship between tables can be expressed via the use of foreign key. By using foreign key, data from the multiple related tables can be retrieved in one PostgreSQL query using the JOIN clause. PostgreSQL join algorithms are similar to MySQL. PostgreSQL's query optimizer is often praised for its sophisticated index usage and advanced optimization techniques. However, the specific behavior and performance of join algorithms can vary based on the database version, configuration, and query complexity. It's important to consider the specific requirements of your application and consult the database documentation for optimal join algorithm selection and performance tuning.

ln Microsoft SQL Server, Adaptive Join is a feature introduced in SQL Server 2017 that enhances the performance of join operations by dynamically selecting the most appropriate join algorithm based on runtime conditions[5]. It is designed to adaptively switch between nested-loop join and hash join during query execution, depending on the characteristics of the data and the available system resources. The Adaptive Join feature allows SQL Server to adaptively choose the join algorithm that provides the best performance for a given query and data distribution. It helps to optimize join performance by dynamically selecting the appropriate join algorithm during query execution, based on actual runtime feedback. Adaptive Join is particularly beneficial in scenarios where the performance of join operations can vary significantly depending on the size and distribution of the data, as well as the available system resources. It helps improve query execution time by leveraging the most efficient join algorithm dynamically.

3 METHODS

3.1 Problems Statements

We aim to enhance the space usage and time consumption in the join process of MySQL. According to their implementation mentioned in section 2.4, it mainly uses nested join, the most basic and ineffective join algorithm. On the other hand, it copies and stores the full tables used in the join process, leading to considerable space waste. To solve these problems and enhance the efficiency of the join process of MySQL, we propose a new join process with three approaches to optimize it. First, there are many unused attributes during the join process. Therefore, we

propose to examine the attributes used in the following operations and retrieve only these attributes to save a bunch of space use. Second, condition matching is vital during the join process, while the current implementation takes much time, especially when two huge tables are joined together. Since we already have the join conditions, we can build a red-black tree for the first table based on the join conditions. The time consumption for matching afterward would dramatically decrease. Third, MySQL copies and stores every selected data. However, because of the concept of the join process, most of the data are duplicated and occupy colossal space. We propose to store the data reference, which takes tiny and fixed space, for the joined table. With these three approaches, the join process's time consumption and space usage can be decreased to an acceptable number.

In the following section, we introduce our system architecture in section 3.2. Then, we explain the implementation and design of the three proposed approaches in sections 3.3-3.5. To better understand our proposed approaches, we use a three-table join SQL statement as an example. The statement can be found in Figure 1. This statement is to count the joined rows that match the given conditions.

3.2 System Overview

Figure 1 shows the workflow of our proposed system. At first, we analyze the necessary attributes for each table. Second, we select the first table with the minimum attributes and build a red-black tree based on the join condition with the second table. Third, we select the second table with the minimum attributes and then search for the matched data on the red-black tree for each row in the second table. The data references for the paired rows would be stored in a data structure for joined table. These two steps would be executed several times. Finally, we would have a joined table, which can be further sent back to the previous steps for another join.

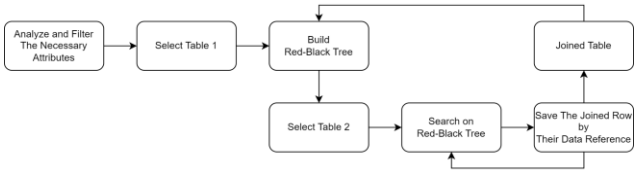

Figure 1: Workflow of the proposed system

3.3 Select and Store The Minimum Attributes

To analyze the selected attributes from the three tables (Samples, r2_func, ida_func), we will focus on the relevant attributes only. Firstly, for the Samples table, we select the attributes ID, Group, Malicious, and Valid (which doesn't need to be stored). Next, we move the join condition "Samples.Valid=True" from the join statement to the Samples table selection. Moving on to the second table, r2_func, we select the attributes ID, SID, and Addr. Then, we perform an INNER JOIN between Samples and r2_func based on the condition Samples.ID = r2_func.SID. Afterward, we select the third table, ida_func, and choose the attributes ID, SID, and Addr.Lastly, we join the resulting table (Samples joined with r2_func) with the ida_func table. Overall, the SQL query combines the selected attributes and performs various count operations, including r2_func_count (count of distinct r2_func IDs), ida_func_count (count of distinct ida_func IDs), and common_records_count (count of distinct r2_func IDs with matching Addr values in ida_func). Samples group the results.ID and Samples.Malicious, and finally ordered by Samples.Malicious and ida_func_count.

```
SELECT Samples.ID,  Samples.Malicious,
    COUNT(DISTINCT r2_func.ID) AS r2_func_count,
    COUNT(DISTINCT ida_func.ID) AS ida_func_count,
    COUNT(DISTINCT CASE WHEN r2_func.Addr = ida_func.Addr THEN
r2_func.ID END) AS common_records_count
FROM Samples
INNER JOIN r2_func ON Samples.Valid=True AND Samples.ID = r2_func.SID
LEFT JOIN  ida_func ON Samples.ID = ida_func.SID
GROUP BY Samples.ID, Samples.Malicious
ORDER BY Samples.Malicious, ida_func_count;
```

Figure 2: Select and Store The Minimum Attributes

3.4 Build Red-Black Tree for Join

By utilizing the concept of a red-black tree to store data, we can optimize the time required for join operations. The traditional approach to join, using nested join, involves combining tables indiscriminately, resulting in a time complexity of O(nm), where n and m are the sizes of the tables being joined.

However, by employing a red-black tree, we can reduce the time complexity to O(nlogm). During the process, we use the join condition as the criterion for constructing the red-black tree. As shown in Figure 3, in our implementation, the join condition is Samples.ID = r2_func.SID. This means that the table Samples is first used to build a red-black tree based on its ID. When joining with r2_func, we can search the red-black tree using SID, allowing us to find the desired data for the join operation in O(logm) time.

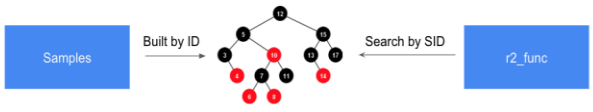

Figure 3: Red-black Tree for Join operation

3.5 Save Data Reference

After determining which tuples were selected based on user-defined conditions, the next step was to extract them and save them

into new tables. However, this process may copy the same data often, increasing unnecessary space consumption. We optimized this process by maintaining their reference (memory address) in a custom data structure. From the below example, we did not extract the selected tuples into a new table and output the additional join table. Instead, we only applied their memory address to record the results from the inner join to the further left join output. The saving reference method reduced the memory requirements and performed efficiently. Moreover, the layered saving process was more flexible, connecting multiple join steps without intermediate tables, as shown in Figure 4.

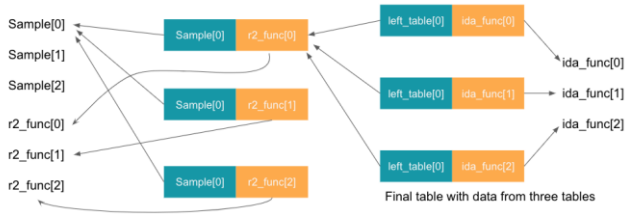

Figure 4: The reference method for reducing the memory requirements

4 PERFORMANCE ANALYSIS

4.1 DATASET

We apply the example in Figure 1 to evaluate the performance of our proposed join process. We have three tables: One "Samples" table contains information about the benign and malicious binaries. The other two tables, "r2_function" and "ida_function," contain function information analyzed by Radare2 and IDA Pro for each binary. For each binary in "Samples," there are different amounts of function information in "r2_function" and "ida_function," and they are connected together by the ID attributes in "Samples" and the SID attributes in "r2_func" and "ida_func." This SQL statement aims to find the count of overlapped and unique functions in "r2_func" and "ida_func," and it is limited to the selected samples recorded in "Samples." We compare the performance between our method and the traditional nested-join method in MySQL from the perspective of time consumption and space use.

Table 1 shows the schema of "Samples," and Table 2 shows the schema of "r2_func" and "ida_func." There are 120887 rows in "Samples," 37320491 rows in "r2_func," and 34605138 rows in "ida_func," respectively. After the first join, there would be 776998 rows in the joined table, and after the second join, there would be 781306132 rows in the final joined table.

Field	Type
ID	int
Hash	char(32)
Path	varchar(512)
Filename	varchar(128)
Size	int
Group	int
Subgroup	int
Malicious	tinyint(1)
PE32	tinyint(1)

Table 1: Table schema of Samples

Field	Type
ID	int
SID	int
Name	varchar(1024)
Addr	bigint unsigned
Size	int

Table 1: Table schema of r2_func and ida_func

4.2 Performance

We evaluate the performance from the perspective of time consumption and space use. For time consumption, it took 10 minutes and 43 seconds to finish the joined statement in MySQL, while our proposed approach takes only 4 minutes and 16 seconds. It saves 60% of the time consumption. On the other hand, for space use, we allocate an independent space for MySQL and analyze the space used during the joined process. Since MySQL has some optimizations to reduce space use, such as discarding unnecessary data after each join process, the space used does not constantly increase. The highest space we recorded is 58 GB, while our proposed approach uses only 23.96 GB in memory. It saves 59% of the space used.

4.3 Space Saved by Data Reference And Filtered Attributes

In this section, we want to know how much space can be saved using data reference and filtered attributes. In our dataset, each row in "Samples" takes 688.25 bytes, and each row in "ida_func" and "r2_func takes 1044 bytes. The space uses are for extreme cases since there are varchar which do not always cost the maximum space if the string length is shorter. In our implementation, we merely select the necessary data. Therefore, each row in "Samples" takes 65 bits, and each row in "ida_func" and "r2_func takes 96 bits. The filtered attributes do save much space compared to storing the whole table.

Another question is how much space can be saved using data reference. The data reference is a fixed value for each data, and in our implementation, we use their memory address, which takes 64 bits, as the saved data reference. In smaller datasets, we can use an array index, such as 32 bits int, as the saved data reference, and it can save more space. The data structure to save the joined table use (n - 1) * size of(data reference) for each row. The variable n here is the count of tables to be joined. Therefore, in our dataset, each filtered row in the final joined table uses 256 bits, while in the copying approach, it would use 257 bits for each row. There is only a slight difference here. However, since the space consumption of our data structure is a fixed value, if we select more attributes, the space used in the copying approach would hugely increase.

5 CONCLUSION

In summary, our research focuses on optimizing the join process by selecting and storing relevant attributes, utilizing a red-black tree for efficient matching, and employing memory-saving techniques. These advancements aim to improve the performance and efficiency of multi-table join operations, especially in scenarios involving extensive datasets. Using the previous command and data as an example, the execution time required by MySQL is 10 minutes and 43 seconds. With our approach, it only takes 4 minutes and 16 seconds to retrieve all the required information. Additionally, our approach also optimizes storage space. The original MySQL implementation required 58GB of temporary storage space, while our proposed method only requires 23.96 GB. This approach eliminates unnecessary data redundancy and conserves memory resources, particularly when dealing with large datasets.

REFERENCES

[1] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, Anastasia Ailamaki. Efficient Massively Parallel Join Optimization for Large Queries. SIGMOD ’22, 12–17 (June, 2022), PA, USA, 122–135. https://doi.org/10.1145/3514221.3517871

[2] Andreas Meister and Gunter Saake. 2020. GPU-accelerated dynamic programming for join-order optimization. (2020). TechnicalReport(2020):https: //www.inf.ovgu.de/inf_media/downloads/forschung/technical_reports_und_ preprints/2020/TechnicalReport+02_2020-p-8268.pdf

[3] Priti Mishra and Margaret H. Eich. 1992. Join processing in relational databases. ACM Comput. Surv. 24, 1 (March 1992), 63–113. https://doi.org/10.1145/128762.128764

[4] David Axmark, Michael Widenius, Jeremy Cole, Arjen Lentz, Paul DuBois, “MySQL 8.0 Reference Manual,” https://dev.mysql.com/doc/refman/8.0/en/.

[5] Woodward, Alden. "Microsoft SQL Server." (2017).

6 WORK DISTRIBUTION

YI-HSIEN CHEN: methodology, Analysis, presentation
YIN-TZU HUANG: abstract, related work
YIH-LIN LIU: related work, presentation
HUI-WEN CHUANG: related work, methodology
PIN-XUAN CHEN: Introduction, dataset

7 VIDEO & CODE

video :

https://www.youtube.com/watch?v=Inw_wWsPWWU

code :

https://gntuedutwmy.sharepoint.com/:f:/g/personal/f08921a05_g_ntu_edu_tw/EhyeD28Irs5Brag5dIgLPk0BTHsSsNMCSp0DREpvfIWxqg?e=46qlCj