Optimizing multiple SQL statements for faster processing

Shashank Prabhakar
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
sxp6288@cs.rit.edu

***Abstract*—**
What started as a contribution towards enhancing the performance of the query optimizer in Apache Calcite [1], which is an actively growing open-source framework for building and managing databases, transitioned into optimization of SQL queries in general. To build an intricate analytic system for any emerging real-world big data application, complex queries are needed. These applications demand very high performance and ultra-practical functionality, which can be is to provide a high-level analysis of the system. These complex queries will have a lot of reusable conditional subexpressions. The idea of reusability can be extended even to big data systems where querying happens in batches and data being dealt with will in terabytes and will be continuously growing. These systems are expected to deliver high performance by processing the queries and obtain results quickly. The main idea behind optimizing these queries would be is to how these conditional sub-expressions can be scrutinized and capitalized upon, which will result in efficient big data systems with reusability. In this paper, the idea of reusable conditional sub-expressions is achieved by building directed acyclic graphs for every sub-expression part of the query and inter-linked accordingly. An optimizer which takes in multiple SQL statements as input will provide an efficient way to enhance the performance of the system to which the SQL queries are plugged into.

***Index Terms*—multi-query optimization; SQL;**

I. Introduction

Big data in the real world involves running a large set of complex queries. These queries will have a common set of conditions or sub-expressions, which can be reused. The optimizer present in the database system may or may not exploit this idea of re-usability. The main idea is to run queries using conditions separately and combine any reusable results. The need to find these common sub-expressions increases, if the querying of a system works in batches. This idea of optimization can be used for streaming and geospatial queries, and semi-structured data queries, where the query calls will be in large and recurring.

The algorithm is based on the idea of representing queries in the form of a DAG [2], which is explained more in detail in section III. The problem of optimizing sets of queries arise when we are dealing with big data analysis. Some of these queries will have some common subexpressions or conditions; this problem is referred to as multi-query optimization, which the paper is dealing with. It can be noted here that common sub-expressions are present even within a single query and can be broken down accordingly into reusable sub-queries. The reusability works on the idea of memoization, where results of expensive query calls can be stored and the same cached result can be used when a query with similar conditions is provided as input.

The results of the optimizer show that the performance is proportional to the number of sub-expressions plugged in with the queries, but a performance degrade can be seen when the queries plugged into the system have no reusable results. Along with the contribution of providing a system which optimizes the performance of multiple queries, the paper also signifies the importance of extracting sub-expressions, which involve operators and other relevant variables, and how they can be reused. The system has been tested with a MySQL system.

II. Motivation

There is always more than one way to fetch relevant results for any query. Identifying the approach is fully down to the optimizer part of the database engine. For the high performance of the database system, it is good to understand the domain of the data that the database engineer is dealing with, before running the queries. Approaches like schema optimization and indexing go hand in hand. Before starting with the design, it is also important to know how the database engine runs a query. In streaming platforms, the database systems are continuously queried to fetch relevant results.

In a scenario like in the X-Ray feature, which is a reference tool present in Amazon prime video, provides general information about the scene. There will be a need to store the characters present in every new scene and the possibility for a character to be recurring in multiple scenes is very high. Repeated calls to the database will be needed if the data is not cached. This problem can be fixed by memoizing the previous query results and stored in a directed acyclic graph, with appropriate linking. This will help in the performance enhancement when dealing with big data will add up, when there are tens of thousands of queries to be made to run in parallel.

III. Design Considerations

The architecture of the system is a simple graph and the flowchart of the data is shown in as shown in Figure 3 The main idea in this project, as mentioned in Section I is to:

- Identify any reusable sub-expressions or conditions if present.
- Recognize any potential results which can be obtained by integrating results of queries together which are already made to run, by building a directed acyclic graph and storing the results in the intermediary nodes.

The DAG built for a batch of queries will be sharing results, by integrating the results stored in the nodes accordingly by proper linkage. Physical properties like sort order are not considered and are not handled in the implementation of the optimizer.

The steps follow one after the other. Initially, the queries which are plugged in, are broken down recursively into sub-queries having just one condition. Information like table name(s), operator(s) and condition(s) are extracted and are stored in the query objects before they are fed into the system for further processing. The idea here is to reduce the number of database scans to a large extent, so as to reduce the read overhead and also save time at the same time, by reusing the already scanned columns from the database. Also, one more careful consideration is that the sub-expressions in a multi-query optimization process cannot always be reused, and the idea here is not a caching system by storing intermediate data always on every scan. This approach can be used on a single large query with a large set of conditional expressions. More about the approach is discussed in depth in the next section.

IV. Approach

Conventional optimizers part of the query processing engine in database systems, do not take into consideration the idea of sharing data amongst the conditional sub-expressions part of the query. The algorithm approach works as follows; Firstly, parse the input SQL statements and build node objects with respect to each query, and extract the necessary details from the queries. Following this, connect the nodes (built query objects) to form a directed acyclic graph, based on the reusability of the results. The built conditional expression DAG will act as a parse tree, by identifying the querying order of the queries. Lastly, Validate the approach, by comparing the results. The complete approach works on the idea of memoization. An overview of the optimizer's algorithm can be seen in Figure 1

To reduce the query load on the database system, once the queries are fed into the system, the query order is set with the following criteria:

- Queries fetching bigger results are given higher priority than others.
- The estimated size of the result is based on the direction in which the mathematical operators in the conditional expression represent. For example, consider a table with 100,000 rows and queries having conditional operators $<$ and $>$ are plugged into the system. The system is designed in a way, so as to, by only parsing the operators and their values, it can prioritize the query order, by identifying which of the queries when made to run first, will result in a data which can be integrated, with the maximum size. If the lesser than operator fetches the bigger result of the plugged-in queries, it will be prioritized to run first. Consider an example:

```
SELECT columns FROM table
WHERE col1 > 200
AND col1 < 1100
```

The query is first broken down into two sub-queries. If the size of the table is 1200, the system would identify to extract sub-expression involving col1 $<$ 1100 first, as it would have a bigger result data intersection, and thus runs the sub-query with this condition. A simple for loop over this result fetching results greater than the condition will get us the final result.

- Query having the most number of sub-expressions/conditions will have the highest weight.

Consider a table with COL3 a column from the table to

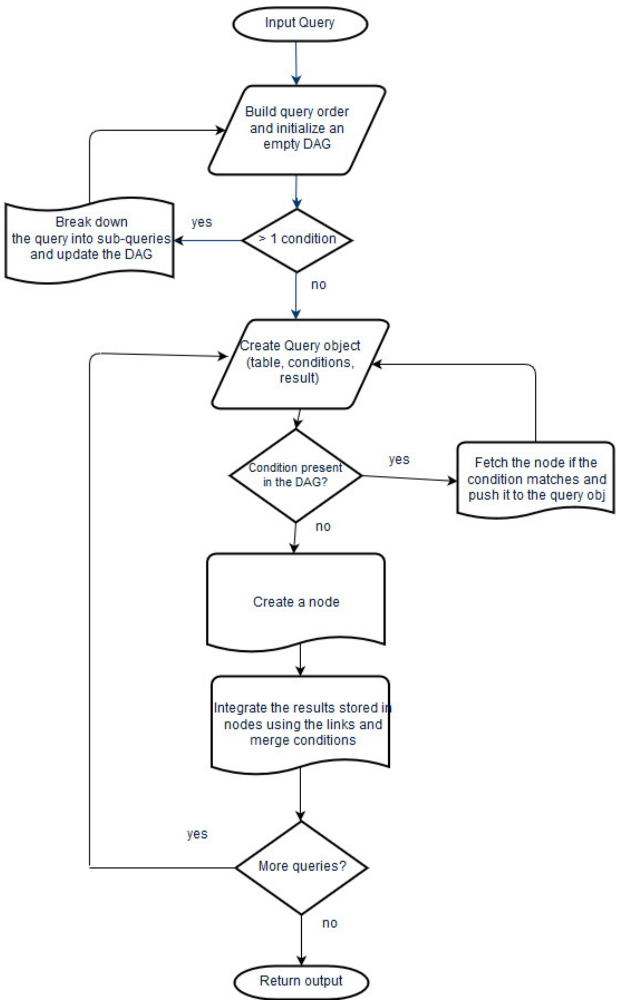

Input Query
Build query order and initialize an empty DAG
> 1 condition
yes
Break down the query into sub-queries and update the DAG
no
Create Query object (table, conditions, result)
Condition present in the DAG?
yes
Fetch the node if the condition matches and push it to the query obj
no
Create a node
Integrate the results stored in nodes using the links and merge conditions
More queries?
yes
no
Return output

Fig. 1. FlowChart

queried has 35 rows; and the database system is plugged in with the following queries:

```
Q1 = SELECT col1, col2 FROM table
WHERE col3 > 10


Q2 = SELECT col1, col2 FROM table
WHERE col3 < 30
```

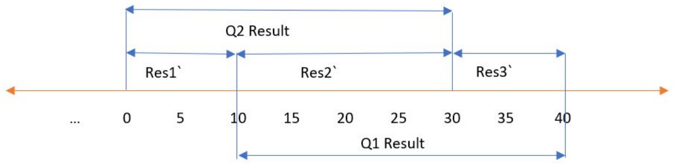


Fig. 2. Result data integration

The idea of re-usability can be understood using the number line diagram as shown in Figure 2. Once the query order is set, Q2 is made to run first, as it fetches bigger of the two results. It can be seen in the figure 2, Q2 result can be split into Res1′ + Res2′, where Res2′ can be extracted from the result obtained after querying the database system by running Q2. Q1 conditions are tested upon Res2 result's conditions, to check for if any data can be reusable. If true, a simple for loop with the conditions present in the subsequent query is made to run on the Q1 result data and Res2′ is fetched. Let's call the data yet to be fetched for Q1 query, Res3′.

To fetch Res3′, a new query is to be made to run, let us call it Q3. The query Q3 generated from the system will be

```
SELECT col1, col2 FROM table
WHERE col3 >= 30
```

The result obtained after running Q3 will be Res3′. Res3′ values are obtained by querying the database for the remaining data, that is values above the limit of Res2′. This would result in the retrieval of a smaller set of items and thus reduce the overhead load of the database query result response. The result of Q1 is obtained by integrating the results together accordingly.

Result of Q1 = Res2′+ Res3′

Consider another set of queries,

```
X = SELECT * FROM table
    WHERE col1 = some_value

Y = SELECT * FROM table
    WHERE col1 = "same_value_as_X"
    AND col2 < 40
    OR col1='some_other_value'
    AND col2 < 40
```

The optimizer will initially identify the query order to be set. As Y will have multiple conditions and more than the number in X, Y will have the highest weight and will be processed first.

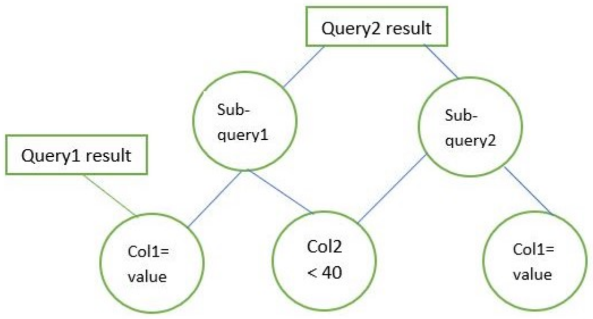


Fig. 3. Resultant graph built

Query2 will be broken down into four different sub-queries, as it has four different conditions present.

```
Y1 '= SELECT * FROM table
      WHERE col1 = "same_value_as_X1"

Y2 '= SELECT * FROM table
      WHERE col2 < 40

Y3 '= SELECT * FROM table
      WHERE col1 = "some_other_value"

Y4 '= SELECT * FROM table
      WHERE col2 < 40
```

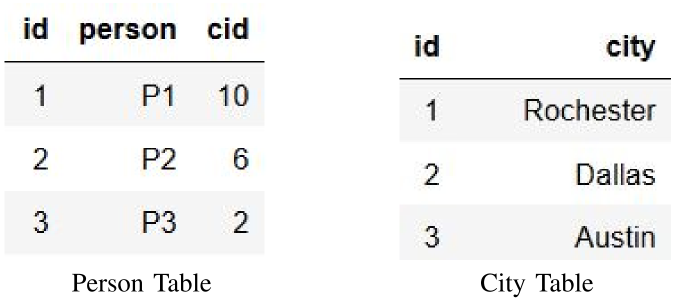
The query order of these four sub-queries are set as well and made to run accordingly. Complex queries involve more than one or two conditions. The idea of reusability was then extended to the construction of directed acyclic graphs, where nodes are accordingly connected based on AND and OR conditions present in the query. The nodes here would contain the condition and the table, on which it is being applied.

The first sub-query will initially peek into the graph to check for the relative table name and conditions that can be reused before querying the database system. As it is empty currently, a node with the table name mapped to the condition is stored with the result fetched from the database system.

Other queries are made to run in the order, the system would have set initially. It can be observed, sub-query with the condition col2 $<$ 40, can be directly fetched from the node have the result for the same.

Edges between the nodes are then set, based on the OR and AND conditions present in the original query. Nodes having the result of col1 = value is intersected and col2 $<$ 40 are joined and the results are intersected to get a new node. This is followed by the union merge of the results of nodes having col2 $<$ 40 and col1 = new_value. The two new nodes formed are then linked together to a new node, where the results stored in the nodes are merged on the union, as shown in the Figure 3.

Consider queries involving multiple tables having joins. Before considering an example, consider a database having two tables, City and Person. The person table has 10 million rows. The head of the tables is as shown in Figure 4.



id	person	cid
1	P1	10
2	P2	6
3	P3	2

Person Table

id	city
1	Rochester
2	Dallas
3	Austin

City Table

Fig. 4. Example tables

```
A = SELECT Person.id, Person.person,
City.city FROM Person
INNER JOIN City
ON Person.cid = City.id
WHERE Person.cid=2
AND Person.id < 10000

B = SELECT * FROM PERSON WHERE cid=2
```

As query B will be fetching the bigger result of the two queries, the optimizer reorders the queries and thus query B is made to run first.

The overall idea is to fetch the values from the table person having city id to be equal to 2. The graph to be constructed will have nodes linked to the graph from the respective tables parsed from the queries. The final result will be a for loop made to run on the results fetched from query 2.

Thereafter, query A will be split into two sub-queries:

```
Sub-query A1 = SELECT * FROM cid=2

Sub-query A2 = SELECT * FROM id < 10000
```

The result of sub-query A1 can be fetched by the already made to run query B. To obtain the result of Sub-query A2, the optimizer will identify that the result can be fetched by running a for loop on the result fetched after querying A1 on the table. Let this result be called A3.

The optimizer then identifies for the join to work, id with the necessary condition from the City table should be obtained and stored in a node A4.

To obtain the final result of query A, a simple intersection of the result of node A3 and A4 will be done, and can be visualized in Figure 5

V. Comparative Analysis

The results of the optimizer's performance on a MySQL database is discussed in this section. The performance results show that the multi-query optimization algorithm works better than the single query optimization when there is a considerable amount of sub-expressions involved which can be reused.

The queries were made to run on a system with the following specs:

- Processor: Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz
- Number of cores: 16

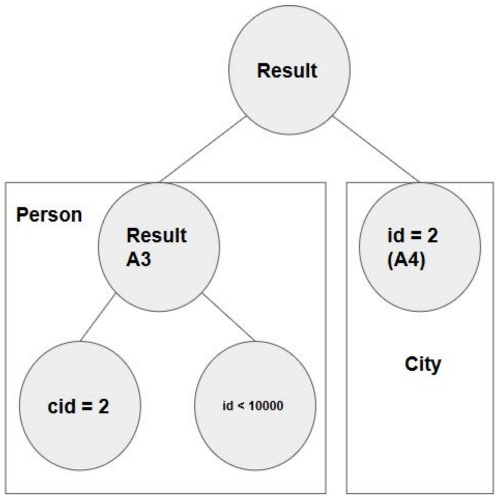


Fig. 5. Join on multiple tables 2

To obtain the performance of the system, different queries that were made to run were:

- Queries with no reusable results:
```
SELECT * FROM Person
WHERE id > 8000000

SELECT * FROM Person
WHERE id < 10000
```
- Queries involving reusable results:
```
SELECT * FROM Person
WHERE id > 600000

SELECT * FROM Person
WHERE id < 8000000
```
- Queries involving reusable results with AND and OR part of the sub-expressions:
```
SELECT * FROM Person
WHERE id > 5000
AND id < 15000

SELECT * FROM Person
WHERE id < 6000
OR cid = 2
```
- Queries number of condition(s), operator(s) and column(s) > 1:
```
SELECT * FROM Person
WHERE id < 50
AND cid = 2

SELECT * FROM Person
WHERE id < 40
```

```
AND cid = 1
```

- Queries having multiple tables:

```
SELECT Person.id,
Person.person,
City.city FROM Person
INNER JOIN City
ON Person.cid = City.id
WHERE Person.cid=2
AND Person.id < 10000

SELECT * FROM PERSON
WHERE cid=2
```

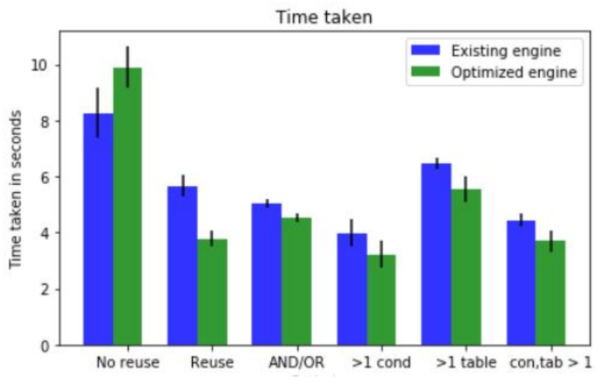


Fig. 6. Bar plot of the performance comparison

- Queries with no reusable results: No performance enhancement is seen nor expected. As the optimizer presumes by default that there are sub-expressions that can be reused and starts breaking down the queries into sub-queries and builds relevant query objects and graph data structure, where some amount of processing time is wasted.
- Queries having reusable results: The optimized version gives results which work faster by 34 % than the MySQL response for the tested queries,
- Queries having AND/OR condition: The optimized version gives results 10.19 % faster on an average run, with error rates of 0.14 seconds and 0.16 seconds respectively.
- Queries having more than one condition: The optimizer gives response 20% faster than the existing MySQL result response.
- Queries having more than one table and one condition only: The optimizer fetches results faster by 14.26% than the existing implementation.
- Queries having more than one table and more than one condition: The optimizer retrieves results faster than the existing MySQL response by 17.91%.

VI. Future Work

Future work involves enhancing the optimizer to support queries, involving more complex conditional expressions. Another add-on can be, the performance of the optimizer should not change, even when there are no reusable conditions. Properties like sort order were not taken into consideration. This is another key factor which is part of the future work.

VII. Conclusion

In this paper, a heuristic graph algorithm was designed for multi-query optimization. The algorithm makes use of the DAG, where every sub-expression will be a node and are inter-linked to form the result of the main query. The implemented optimizer edges out the existing MySQL implementation only when reusable conditions are part of the sub-expression/condition part of the queries. In conclusion, a practical groundwork was laid out for multi-query optimization and can be extended for other database systems.

Acknowledgment

The author would like to thank Dr. Michael Mior for providing continuous feedback and lending a helping hand in the entirety of the completion of the project.

References

[1] E. Begoli, J. Camacho-Rodrguez, J. Hyde, M. J. Mior, and D. Lemire, “Apache calcite: A foundational framework for optimized ...” [Online]. Available: https://arxiv.org/pdf/1802.10233.pdf

[2] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, “Efcient and extensible algorithms for multi query optimization.” [Online]. Available: https://www.cse.iitb.ac.in/ sudarsha/Pubs-dir/mqo-sigmod00.pdf