Multi-query Optimization

Abhigna Banda IMT2018002

***Abstract*—AIR is a big data analytic framelike similar to Apache Flink, Apache Spark. There are many query optimization techniques for Apache Flink, Apache Spark. We try to use those techniques and some more optimized techniques for multi query optimization on AIR. In this paper, we discuss the algorithm for multi-query optimization. We also discuss the implementation details of the algorithm. We will see various use cases where the algorithm works. We take a small dataset and observe the experimental results and compare them in this paper.**

***Index Terms*—big data, Flink, AIR, batch processing, stream processing, multi-query optimisation, MQO, MR share and relaxed MR share**

I. Introduction

AIR is a distributed stream processing engine. It is written in C++. It uses concepts like Message Passing Interface, pthreads for multithreading. It is directly deployed on top of a common HPC workload manager such as SLURM. AIR implements a light-weight, dynamic sharding protocol (referred to as ”Asynchronous Iterative Routing”), which facilitates a direct and asynchronous communication among all worker nodes and thereby completely avoids any additional communication overhead with a dedicated master node. AIR is currently capable of processing complex dataflows, consisting of directed-acyclic graphs (DAGs) of streaming operators, in a distributed and highly multithreaded manner.

There are a lot of algorithms that do multi-query optimization for the big data analytic frameworks like Apache Flink, Apache Spark. For example in Flink we have a Join-MOTH system (J-MOTH) that is proposed to exploit sharing data granularity, sharing join granularity, and sharing implicit sorts within multiple join queries. For sharing data, Multi Query Optimization using Tuple Size and Histogram (MOTH) system, has been introduced to consider the granularity of sharing data opportunities among multiple queries. For sharing sort, Sort-Based Optimizer for Big Data Multiquery (SOOM), has been introduced to consider the implicit sorts among join queries. For sharing join, additional modules have been tailored to the J-MOTH optimizer to optimize sharing work by exploiting shared pipelined multiway join among multiple multiway join queries. In Flink, we also have a Flink query optimiser that translates the queries into jobs which are repeatedly submitted with similar tasks.

In one of the papers (1), a MQO algorithm has been proposed on Apache Flink. The Flink-MQO system is a software component which is built on top of Flink and implements two existing works - the MRShare and relaxed MRShare techniques.

MRShare is a concurrent sharing framework where I/O cost is dominant (Nykiel et al., 2010). So, it considers the sharing opportunities of scans, map output and map functions sharing. However, the relaxed MRShare relaxes and generalizes the overlapped queries to increase sharing opportunities over a single job (Wang and Chan, 2013). According to relaxed MRShare, the shared map input scan and map output have been studied and algorithms have been introduced to select a computation plan for a batch of jobs in the MapReduce context. Also, additional optimisation techniques (i.e GGT and MT) have been introduced in the relaxed MRShare. Moreover, a comparative study of MRShare and relaxed MRShare techniques using predicate-based filters on MapReduce is introduced in Sahal et al. (2016). According to the comparative results, it is found that the relaxed MRShare technique outperforms MRShare technique with respect to query execution time for shared data regarding predicate-based filters in MapReduce.

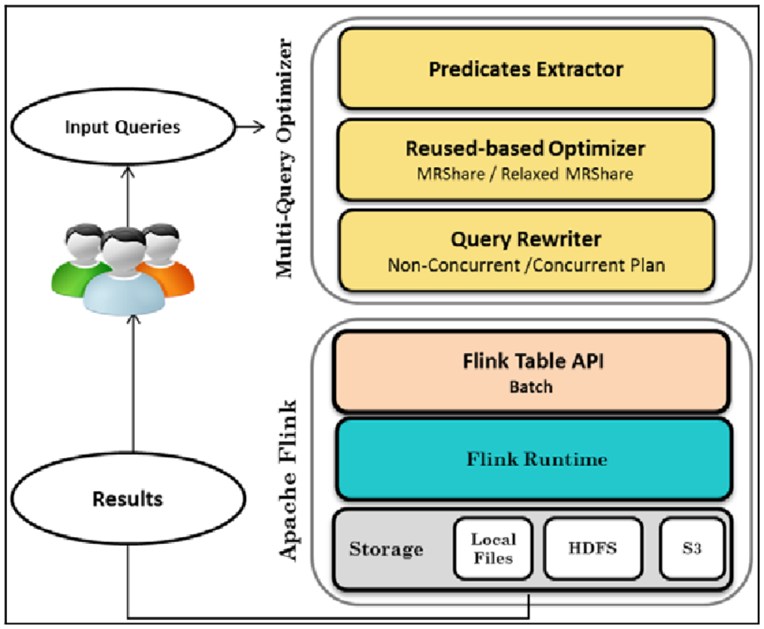

Fig. 1.

The proposed Flink-MQO system consists of two layers: multi-query optimiser and the underlying Apache Flink. The multi-query optimizer has the following algorithm:

(i) Predicates extractor module : The predicates extractor module parses the queries in terms of tokens (i.e., SQL commands). Then, it extracts the pre-defined selection predicates such as BETWEEN operator and AND and OR operators from WHERE clause.

(ii) Reused-based multi-query optimiser module (MRShare/Relaxed MRShare) : The plan schedules the queries in such a manner that shared selected predicates are computed once. It saves them temporarily and they are reused immediately by subsequent queries.

Set of predicates TO Set of plain queries TO Set of ordered queries in DAG form

In relaxed MRShare query, smallest overlapped predicate is chosen

(iii) Query rewrite module (non-concurrent multi-query plan and concurrent multi-query plan): Non-concurrent queries are executed sequentially from root to few parents.

Concurrent queries are queries executed simultaneously. Leaf queries can reuse non-concurrent queries result + non-shared queries that are listed in concurrent queries list.

This algorithm is applied on top of Apache Flink. The Flink-MQO system submits the optimized multi-query execution plan to Apache Flink. The Flink Table API translates the queries within the optimized multi-query plan into a set of relational operators such as selection predicates. These operators are scheduled into a set of batch processing jobs which are submitted to the Flink runtime environment. Flink reads the data from big data infrastructure and performs the optimized jobs over the lower data size comparable with the native processing. The final desired outputs are sent back to the big data analysts.

II. Algorithm

Our algorithm aim is to group overlap queries, find the optimized query and schedule them so that redundancy is reduced.

When a list of queries are sent, the following process is performed on them.

- We take each query and check if it is a declarative statement. If it is a declarative statement, we just add them to a list that contains all declarative statements.
- If it is not a declarative statement, we check if the statement has any keywords such as “ORDER BY”, “GROUP BY”, “HAVING”. If so, we just remove that part from the query. We use “ORDER BY” to order the queries. For finding overlapping queries it is not helpful so we remove that part from the query. “GROUP BY” is also performed to group the queries which is again not useful for finding the overlapping part in queries so we remove that part also.
- We then take each query and check if it has a “WHERE” condition. If there is no where condition we cannot form groups according to our algorithm, so put aside all the queries that have no “WHERE” condition.
- Now each query is checked for keywords like “AND”, ”OR”, “NOT”, “BETWEEN”, “IN”, “NOT IN”. If there are such keywords, it means that we can divide the query into predicates.

  Each input query is modeled as (R, A, P) where R is the relation names (i.e., table name(s)), A is the set of attributes and P is the selection predicates filter which is applied to retrieve the desired output. Consequently, the shared selection predicates (SSP), could be identical or subset which can be defined by the following definition:

  Definition 1: Shared selection predicates (SSP)

  Qi,Qj (Ri Rj Ai Aj PiPj)$\rightarrow$SSP(Qi,Qj)
- After all the queries are converted to predicates, they are put into groups. This grouping can be done in many ways but we have done it in the following manner.

  (i) Filter-1 : Based on table name

  We group the predicates based on the table names. While parsing we look for the format (from _,_ or from _,_ as _,_ or similar to these syntaxes) and get the list of tables. First grouping is based on the number of tables. Number of groups/lists formed = no of tables.

  (ii) Filter-2 : Based on attributes/condition

  We take each group and further divide them. We check out for the “where” condition. Based on the ‘where’ condition we classify them into groups. We take the queries that have the same ‘where’ condition (same attribute) and form groups. We call the attribute after the “WHERE” condition as the primary attribute.

  (iii) Filter-3 : Based on the operator

  Here groups are formed based on the operator (‘¿’ and ‘¡’).
- After all the grouping is done, we take each group and look out for that value of the primary attribute and arrange each group of queries in ascending or descending order
- We go through each group and get the attributes after the “SELECT” part. We call these attributes secondary attributes. For each group, we find the list of distinct secondary attributes.
- For each group, we then form a query with the list of distinct secondary attributes and extreme(either first or last as they group has been arranged in ascending or descending order) value of the primary attribute. We call this query a Super query. For each group, we get a super query.

Finally all these super queries are computed first and the output of them is used in execution of other queries.

III. Implementation details

The entire code for optimization of queries is written in Java. Each sql query is written in a .txt file and are taken as inputs as the arguments while executing the code. The sql queries are executed in mysql.

We can implement the algorithm in a simple way or we can use the object concept. We have implemented the algorithm using the object concept.

Here also we have 2 ways:

Option 1: We can simple apply the same algo and use object concept

Option 2: We can slightly modify the algo while using object concept.

The different objects existing are Query, Select, From, Where.

Query has Select, From, Where as objects as properties.

Select objects will have all the secondary attributes of a query as properties.

From object will have all vector of table names

Where object will have vector of primary attributes

■ Option 1 in detail : After filtering is done, groups are in the form of lists. We then simply find the super query.

■ Option 2 in detail : After filtering is done, we give ranks for grouping purpose. This rank can be given while execution of the queries or they can be predefined.

(a) Predefined

We have numbers for the table names, primary attributes and operator predefined. Helpful for execution of large number of queries.

We have txt files (tablenameMap.txt, primaryAttributesMap.txt, operatorsMap.txt) that shows us the mapping of the attributes and the numbers.

The query has extra properties like f1r, f2r and f3r that says filter1rank and so on.

While filtering, make the objects as well give the ranks simultaneously.

After all the filters are done.

Go through all the queries and make lists of all queries that have the same rank. (here u can sort them for easy grp)

After making groups of all such queries, make superQuery for each group and store them in a list

The list of superQueries is just list of strings and not actual queries

(b) spontaneous

We give numbers to the attributes, table names as we encounter them. Rest all is the same as the Predefined one. Useful in case of execution of a small number of queries.

The following things are taken care of by the algorithm.

■ queries from same table are formed as a group and a super query is found

■ queries with different operators are separated and formed as different groups

■ queries with different primary attribute are formed as groups

■ alias are taken care : alias for secondary attributes and alias for table names

■ when there is no where condition in the sql queries, all of such queries are made into a list and just passed, no super queries are formed for it

■ when there are AND and OR keywords involved, the query is split into 2 predicates

■ when there is a NOT in the query, the query is split into 2 predicates when only that value is excluded

■ the BETWEEN keyword is replaced with included AND

■ IN is replaced with OR

■ NOT IN is replaced with !OR

■ ORDER BY, GROUP BY, HAVING are ignored and only the part till where condition is taken care

■ the declarative sql statements are left as they were

A. Schema

The schema is as follows :

Student

Student(ID int, Name varchar(30), Age int, Weight double, GPA double);

Instructor

Instructor(ID int , Name varchar(30) , Age int, Weight double, Dept varchar(30) , Salary double);

Course

Course(ID String , Title String , Semester int);

We have 3 tables namely Student, Instructor and Course. All the queries are executed on this dataset.

The creation of tables and inserting data into tables is written in mysqlDBStatements.txt file.

B. Examples of queries with their performance time

We have categorized the queries into 10 sets. Each set serves a purpose and has a number of queries. Each set covers all the queries that could be possible for that particular sql type query.

Each query is a separate .txt file. All the queries of all sets are written in queriesList.txt file.

To check how good is our algorithm, we find the execution time of each set of queries normally and compare it with the execution time of set of queries that are executed with the algorithm.

We find the execution time of queries using sql statements like

1) SET profiling=1;
2) Query1
3) Show profiles

Example:

1) SET profiling=1;
2) SELECT Name, Age, Salary FROM Instructor WHERE Age $\geq$ 30
3) Show profiles

'show profiles' command gives us the execution time of each query. We call this as 'time1'

We find the time taken for algorithm execution and add it to the execution time of query when algorithm is applied. We call this as 'time2'

We compare 'time1' and 'time2' and observe the results. The results might not be satisfying as we are taking a small set of data. But 'time2' will be definitely less than 'time1' for large datasets.

The 10 sets attached as 10pdf's and the observations of the sets are as follows :

1) SET 1
   When Q1 to Q13 are passed,
   (Q1, Q2, Q3, (SELECT Age, Name, Salary FROM Instructor WHERE Age $\leq$ 70))

(Q4, Q5, Q6, Q7, (SELECT * FROM Student WHERE Age $\gg$ 40))

(Q8, Q9, Q10, (SELECT Age, Name, Weight FROM Student, Instructor WHERE Age $\leq$ 70)

(Q11, Q12, Q13, (SELECT Age, Name, Salary FROM Instructor WHERE Weight $\leq$ 70))

When Q1 to Q21 are passed,

(Q1, Q2, Q3, Q14, Q15, Q16, Q17, (SELECT Age, Name, Salary FROM Instructor WHERE Age ≤ 100))
(Q4, Q5, Q6, Q7, (SELECT * FROM Student WHERE Age $\gg 40$))
(Q8, Q9, Q10, Q18, Q19, Q20, Q21 (SELECT Age, Name, Weight FROM Student, Instructor WHERE Age ≤ 90)
(Q11, Q12, Q13, (SELECT Age, Name, Salary FROM Instructor WHERE Weight ≤ 70))

2) SET 2
When Q1 to Q4 are passed,
(Q1, Q2, Q3, (Q1, Q2, Q3))
3) SET 3
When Q1 to Q3 are passed,
(Q1, Q2, Q3, (SELECT Age, Dept, Name, Salary FROM Instructor WHERE Age is more than 50 and SELECT Age, Dept, Name, Salary FROM Instructor WHERE Age ≤ 60))
4) SET 4
When Q1 is passed,
(Q1, (SELECT Name, Age, Salary FROM Instructor WHERE Age is less than 30 and SELECT Name, Age, Salary FROM Instructor WHERE Age ≥ 30))
5) SET 5
When Q1 is passed,
(Q1, (SELECT Name, Age, Salary FROM Instructor WHERE Age is more than 20 and SELECT Name, Age, Salary FROM Instructor WHERE Age ≤ 30))
6) SET 6
When Q1 is passed,
(Q1, (SELECT Name, Age, Salary FROM Instructor WHERE Age = 10 and SELECT Name, Age, Salary FROM Instructor WHERE Age = 20 and SELECT Name, Age, Salary FROM Instructor WHERE Age = 30)
7) SET 7
When Q1 is passed,
SELECT Name, Age, Salary FROM Instructor WHERE NOT Age = 10
SELECT Name, Age, Salary FROM Instructor WHERE NOT Age = 20
SELECT Name, Age, Salary FROM Instructor WHERE NOT Age = 30
To
SELECT Name, Age, Salary FROM Instructor WHERE Age less than 10 AND Age more than 10
SELECT Name, Age, Salary FROM Instructor WHERE Age less than 20 AND Age more than 20
SELECT Name, Age, Salary FROM Instructor WHERE Age less than 30 AND Age more than 30
(Q1, (SELECT Name, Age, Salary FROM Instructor WHERE Age is less than 30 and SELECT Name, Age, Salary FROM Instructor WHERE Age is more than 10)
8) SET 8
When Q1 to Q6 are passed,
(Q1, Q2, Q3, Q4, (Q1, Q2, Q3, Q4))
(Q5, Q6, (SELECT Name, Age FROM Instructor where Age ≥ 20))
9) SET 9
When Q1 to Q3 are passed,
(Q1, Q2, Q3, (Q1, Q2, Q3))
10) SET 10
When Q1 is passed,
(Q1, (Q1))

For set1 when we send all the 21 queries, total time1=0.047983 And total time2=0.01767+0.022 = 0.02967

We see that time2 is less than time1, so our algo is efficient.

For set2 when we send all the 3 queries, total time1=0.002741 And total time2=0.002741+0.013 = 0.014741

We see that time2 is more than time1, this might be because we are dealing with small no of queries.

For set3 when we send all the 3 queries, total time1=0.004925 And total time2=0.004593+0.015 = 0.016593

We see that time2 is more than time1, this might be because we are dealing with small no of queries.

For the set4, set5, set6, set7 , when we just compare the time1 with t, t is less than time1 but when we add the algo time then time2 is more than time1. As we are doing for just one query, time1 is less than time2 but if we do for many no of queries time2 can be less than time1.

We observe that only for set1, time1 is more than time2 and for all the other sets, time2 is more than time1. It may mean that our algo is efficient for large no of queries.

IV. Conclusion

In this paper, we have applied an algorithm for multi-query optimization. We see that our algorithm works well for large no of queries. Here large no of queries could be 100 queries also. The algorithm can be easily implemented. This algorithm is applied on top of AIR. From experimental results, we can prove that the algorithm is sound. We have covered a lot of use cases but there might still be some use cases that we have not encountered. This algorithm has a good scope and can be used for multi-query optimization.

V. Acknowledgement

I would like to thank Professor Vinu E. Venugopal for giving me guidance and support throughout the project. It was a great learning experience for me.

References

[1] https://ieeexplore.ieee.org/document/9235069
[2] https://ieeexplore.ieee.org/document/9235069
[3] https://dl.acm.org/doi/fullHtml/10.1145/3493700.3493758BibPLXBIB0005
[4] https://dl.acm.org/doi/pdf/10.1145/3371158.3371206
[5] https://arxiv.org/pdf/1802.08496.pdf
[6] http://dcsg.cs.umn.edu/Papers/albert/Sana.pdf
[7] https://core.ac.uk/download/pdf/82403072.pdf
[8] Krishnamurthy S., Wu C., and Franklin M. On-the-fly sharing for streamed aggregation. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2006, pp. 623–634.
[9] Exploiting Sharing Join Opportunities in Big Data Multi Query Optimization with Flink: https://www.hindawi.com/journals/complexity/2020/6617149/
[10] Multi-Query Optimization in Wide-Area Streaming Analytics : http://dcsg.cs.umn.edu/Papers/albert/Sana.pdf
[11] Sketch-Based Multi-Query Processing over Data Streams : https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73.3775rep=rep1type=pdf https://dl.acm.org/doi/pdf/10.14778/3372716.3372718
[12] Apache flink implementation : https://www.tutorialspoint.com/apache_flink/apache_flink

[13] https://blog.knoldus.com/flink-union-operator-on-multiple-streams/
[14] https://stackoverflow.com/questions/40452368/multiple-streams-support-in-apache-flink-job
[15] https://medium.com/@knoldus/flink-join-two-data-streams-1cc40d18a7c7
[16] https://www.alibabacloud.com/blog/basic-apache-flink-tutorial-datastream-api-programming_595685
[17] https://medium.com/@mustafaakin/flink-streaming-sql-example-6076c1bc91c1
[18] https://gist.github.com/mustafaakin/457859b8bf703c64029071c1139b593d
[19] https://github.com/fhueske/flink-sql-demo
[20] https://github.com/ververica/flink-sql-cookbook
[21] https://www.irjet.net/archives/V8/i11/IRJET-V8I1119.pdf
[22] https://downloads.hindawi.com/journals/complexity/2020/6617149.pdf
[23] https://www.ijeat.org/wp-content/uploads/papers/v9i2/B3081129219.pdf
[24] http://article.nadiapub.com/IJGDC/vol9_no5/20.pdf
[25] https://stackoverflow.com/questions/13767761/how-to-dynamically-create-lists-in-java
[26] https://stackoverflow.com/questions/4805606/how-to-sort-by-two-fields-in-java With strings there can be ¡, ¿, ¡=,¿=
[27] https://www.educba.com/sql-compare-string/ Mysql references
[28] https://stackoverflow.com/questions/10577374/mysql-command-not-found-in-os-x-10-7
[29] https://stackify.com/heres-how-to-calculate-elapsed-time-in-java/