Turkish Journal of Electrical Engineering and Computer Sciences

Volume 26 | Number 5 Article 27

1-1-2018

Optimization in the catalyst optimizer of Spark SQL

MEENU CHAWLA

VINITA BANIWAL

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 2489 – 2499

doi:10.3906/elk-1707-6

Optimization in the catalyst optimizer of Spark SQL

Meenu CHAWLA, Vinita BANIWAL*
Department of Computer Science & Engineering, Maulana Azad National Institute of Technology, Bhopal, India

Received: 02.07.2017 • **Accepted/Published Online:** 09.05.2018 • **Final Version:** 28.09.2018

Abstract: Apache Spark is one of the most technically challenged frameworks for cluster computing in which data are processed in a parallel fashion. The cluster consists of unreliable machines. It processes a large amount of data faster compared to the MapReduce framework. For providing the facility of optimized and fast SQL query processing, a new unit is developed in Apache Spark named Spark SQL. It allows users to use relational processing and functional programming in one place. It provides many optimizations by leveraging the benefits of its core. This is called the catalyst optimizer. This optimizer has many rules to optimize queries for efficient execution. In this paper, we discuss a scenario in which the catalyst optimizer is not able to optimize the query competently for a specific case. This is the reason for inefficient memory usage and increases in the time required for the execution of the query by Spark SQL. For dealing with this issue, we propose a solution in this paper by which the query is optimized up to the peak level. This significantly reduces the time and memory consumed by the shuffling process.

Key words: Shuffling, pushdown filter, rules, joins, catalyst optimizer

1. Introduction

In the past few years, many frameworks have been introduced to deal with unstructured and massive data. Every framework contributed some major functionalities for the better processing of enormous collections of data. Similarly, a new framework has also been introduced by the Apache foundation, which is Apache Spark [1]. Spark is responsible for cluster computing and processes applications by data-parallel computation on a cluster of nodes.

A new module has also been introduced in Apache Spark, which is called Spark SQL [2]. It allows Spark programmers to utilize the benefits of the complex analytics libraries of Apache Spark. This module combines relational processing and functional programming APIs of Spark so that programmers can easily switch between both mechanisms and do not have to think about choosing one facility at a time. The following contributions have made it possible to achieve both functionalities in one module. First is the catalyst optimizer, which is the core of Spark SQL and is extendable so that new rules can be added easily as and when the need arises. The second is DataFrame, which is a component of Spark. The performance and working of Spark are considerably different from that of MapReduce [3]. It has the facility to evaluate operations in a lazy manner so that it can perform relational optimizations. Spark SQL is designed in such a way that it can handle structured data in an efficient manner.

A midquery fault tolerance mechanism is provided by Spark SQL by using an abstraction called a resilient distributed dataset (RDD) [4]. The RDD is the key reason to choose a Spark framework for interactive and

*Correspondence: vinitabaniwal@live.com

iterative applications because these applications require reusing the data and RDDs make it possible to reuse the data efficiently by keeping the intermediate data in the memory.

In this way, Spark provides numerous advantages over the previously introduced frameworks. Spark SQL has a catalyst optimizer in its core. This optimizer has several rules for optimizing relational queries. These rules are applied to a data structure called a tree. The tree is the main data type in this optimizer. Before executing a query, it passes through four phases of the catalyst optimizer. These phases are analysis, which is applied to either SQL queries or DataFrame; logical optimization, which is done on the analyzed logical plan; physical planning, during which the optimized logical plan is taken as input and some physical operators are applied to optimize the plan; and code generation, in which the physical plan is converted into Java bytecode to run on each node of the cluster.

In this manner, relational queries are executed by applying some richer optimizations on them. We considered the physical plans of specific queries and observed that some optimizations are still possible to be applied before executing the queries. After applying all possible optimizations manually on those queries, we saw that the time and memory required by the shuffling process are significantly less for queries on which manual optimization has been done, compared to the original query. Thus, there is a need to add this optimization to the catalyst optimizer so that unnecessary memory consumption and time can be reduced to a certain extent. To overcome this issue, we designed an algorithm. When these queries pass through the algorithm the memory and time consumption get reduced during the shuffling process.

2. The existing method to optimize queries having joins and filters

In this paper, we discuss a scenario in which some more optimizations are still possible to be imposed on queries, which are currently not applied by the corresponding rule of the catalyst optimizer. This specific scenario occurs when a query has joins and filters in it. Before understanding the query execution, let us understand the shuffling process, which is the key subject of this study.

2.1. Shuffling

Shuffling [5] is a process that is part of the Spark driver. Shuffling is an intermediate step of the Spark application that consumes resources and can affect the process of Spark applications because it requires disk I/O, serialization, and network I/O to calculate the result. It takes place when a Spark job contains data aggregation operations like reduceByKey, groupByKey, or Join.

Shuffling is a process in which data are redistributed over partitions and transferred between stages. To organize all the data for a reduce task, Spark needs to perform an all-to-all operation, which is called shuffling. The shuffle phase is a combination of partitioning data and aggregating data according to the situation. During this process, all partitions are read to discover the values related to all the keys. These values are used to calculate the result for each key. During this process, Shuffle Write and Shuffle Read operations take place separately. These operations are shown in Figure 1.

"Shuffle Write" can be assumed as the sum of all written serialized data on all executors before transmitting, normally at the extremity of a stage. "Shuffle Read" is the total amount of reading of serialized data on all executors at the starting of any stage. All the results related to single map tasks are kept in memory. Later, these results are sorted according to the target partition and then written to a single file. If the data are too large for the memory, then Spark starts to spill these tables to the disk, incurring additional overhead of disk I/O and increasing garbage collection. Shuffling is basically a process to maintain a shuffle file for each partition.

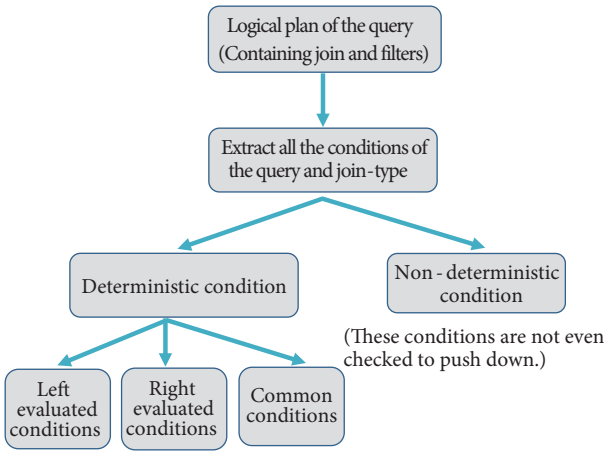

Figure 1. Method of selecting the filters to be pushed by existing rule.

Normally in Spark, the number of shuffle files generated during shuffling is M × R.

Here, M = total no. of map tasks and R = total no. of reduce tasks.

Shuffling is improved in Spark by using the sort-based [6] technique for particular scenarios. For understanding the optimizations that are of interest to us, let us consider the following queries. Two queries are written here. In both the queries, some conditions and joins are placed and both queries have the same objective and produce the same result after execution.

Query1:

Select 2_col2 from 1_table join 2_table
Where (1_col1=2_col1
and 1_col2="ABCD" and 1_col3="EFGH"
and 1_col4="1111" and 2_col3="2222"
and 2_col4="3333" and 2_col5="WXYZ")
OR
(1_col1=2_col1
and 1_col2="aaaa" and 1_col3="bbbb"
and 1_col4="1111" and 2_col3="2222"
and 2_col4="3333" and 2_col5="cccc")

Query2:

Val 1_half = hiveContext.sql ("select 1_col1, 1_col2 from 1_table
Where (1_col2="ABCD" and 1_col3="EFGH" and 1_col4="1111")
OR
(1_col2="aaaa" and 1_col3="bbbb"
and 1_col4="1111") ")
Val 2_half = hiveContext.sql ("select 2_col1, 2_col2, 2_col5 from 2_table
Where (2_col3="2222" and 2_col4="3333"
and (2_col5="WXYZ" or 2_col5="cccc") ")
1_half.registerTempTable ("1_filter")

```
2_half.registerTempTable ("2_filter")
hiveContext.sql ("select 2_col1 from 1_filter join 2_filter
Where (1_filter.1_col1=2_filter.1_col1
and 1_filter.1_col2="ABCD"
and 2_filter.2_col5="WXYZ")
OR
(1_filter.1_col1=2_filter.1_col1
and 1_filter.1_col2="abcd"
and 2_filter.2_col5="cccc") ")
```

After executing both the queries on Spark SQL, the memory required by both shuffle write and shuffle read operations is calculated and the overall time required to process the queries is noted in order to compare the queries. The results obtained by executing these two queries are shown in Figures 2 and 3.

Stage ID	Duration (sec)	Input	Shuffle-Read	Shuffle-Write
5	2		34.1 KB	
2	0.3		256.0 B	
1	48	1808.5 MB		34.1 KB
0	276	10.6 GB		3.2KB

Figure 2. Execution of Query 1.

Stage ID	Duration (sec)	Input	Shuffle-Read	Shuffle -Write
5	1		6.6 KB	
2	0.3		32.0 B	
1	49	1808.5 MB		3.4 KB
0	288	10.6 GB		3.2 KB

Figure 3. Execution of Query 2.

As we can see, in query 2, some manual optimizations are done by pushing all possible filters before applying a join operation; therefore, query 2 takes a very smaller amount of time and memory during shuffling for its execution. In query 1, only existing optimizations are applied, and some filters are pushed before joining, so it consumes a lot of memory and time compared to query 2.

There is a dedicated rule of the catalyst optimizer for pushing the filters and evaluating them before joining operation. The rule is "Push Predicate Through Join", which is applied to queries that have "joins" and "filters". It has its own criteria to select filters, which can be pushed before join, and it does not care about the remaining filters. When this existing rule is applied to query 1 and query 2, a significant decrease in time and memory consumption is observed.

Let us understand the criteria used by this rule for the selection of the filters to be pushed. When the query has some groups of filters, where each group is separated by the "or" operator and filters of each group are separated by the "and" operator,

- This rule takes the logical plan of a query as an input and classifies the filters into deterministic and nondeterministic filters.
- The filters that are deterministic in nature are checked for selection to be pushed.
- The deterministic filters are then segregated into Left-Evaluated Filters, Right-Evaluated Filters, and Common Conditions. Only those filters having the same column name and same literal value are pushed, which are present in all the groups of filters.

1. Left-Evaluated Filters are the filters that are pushed before join to the left side table of the join operation.
2. Right-Evaluated Filters are filters that are pushed before join to the right side table of the join operation.

3. Common Conditions are the filters that remain after the above segregation. They are evaluated after join operations and they are the reason for unnecessary memory and time consumption during shuffling. This functionality is illustrated by Figure 4.

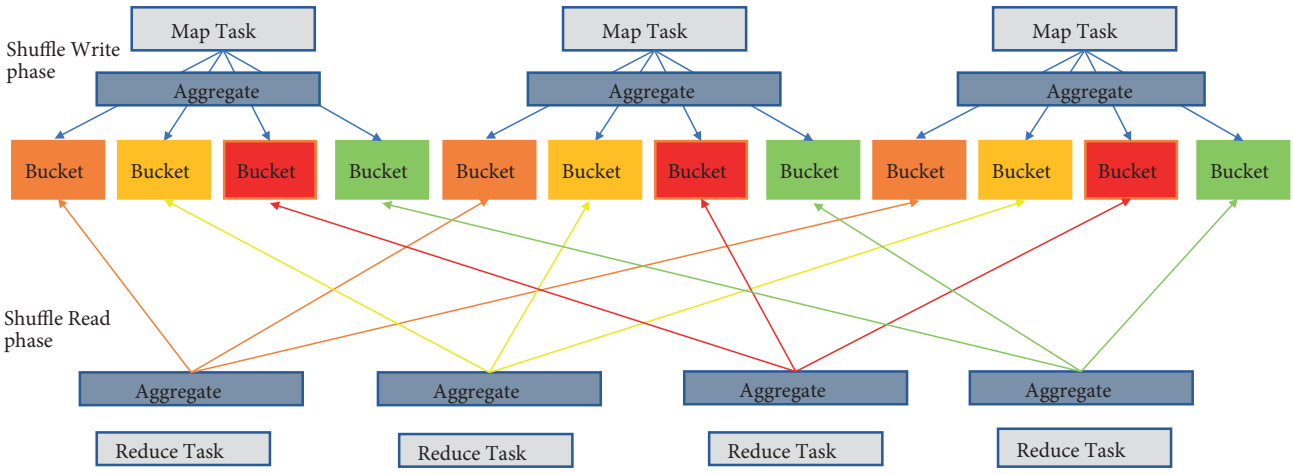

Figure 4. Shuffle read and shuffle write.

In this way, from Query 1, 1_col4="1111" and 2_col3="2222" and 2_col4="3333" are pushed before the join operation because these filters are part of both the groups separated by the "OR" operator. The remaining filters are not pushed.

Still, some of the remaining filters can be pushed before the joining operation, which will result in an efficient shuffling process. Proof of this is shown in query 2, where all filters are pushed before join and evaluated in a different way. Therefore, the memory consumption and time required by query 2 are much lower compared to query 1. This is depicted in Figure 2.

Hence, there is a need to add an algorithm that can deal with the above-mentioned issue. This issue is also discussed in Apache Jira.

3. Proposed solution

The purpose of providing an additional optimization to the query is to enhance the performance of the shuffling process during execution. It has been elucidated above that shuffling is already a costly operation because it transfers data from one executor to another.

To optimize the query, the following algorithm has been designed:

Here, Step 5 is the most essential as it reduces the size of the data processed during shuffling. Step 6 is executed as a cross-check to compute the final accurate result. This is done because there may be some data from Step 5 that may not actually be part of the final result.

4. Results and analysis

For proving the efficiency of the proposed algorithm, some queries are designed. They are executed by both the existing and proposed approaches. This clearly demonstrates the benefits of the proposed approach.

Algorithm

1: Extract the Common Conditions from the query, which are not pushed by the existing system.
2: Separate the filters as Left-Evaluated Filters and Right-Evaluated Filters.
3: Prepare a Left Group by applying the "OR" operator between all Left-Evaluated Filters when considered pairwise.
4: Prepare a Right Group by applying the "OR" operator between all Right-Evaluated Filters when considered pairwise.
5: Push the Left Group to the left-side table of join and the Right Group to the right-side table of join to evaluate filters before join.
6: Feed the Left Group and the Right Group into the query even after the join operation for ensuring the accuracy in calculation of the result.

Query 1:

Select count (*) FROM table1 join table2
|where (table1.User_Name=table2.User_Name
|and table1. 'From'='From: richard.tomaski@enron.com'
|and table1.Subject='Subject: Harper Deals'
|and table1.Mime_Version='Mime-Version: 1.0'
|and table2. 'Date'='Date: Tue, 3 Apr 2001 '
|and table2.Message_ID='Message-ID:<12345>')) | or
|(table1.User_Name=table2.User_Name
|and table1. 'From'='From: laura.vuittonet@enron.com'
|and table1.Subject="Subject: Tony's deals"
|and table1.Mime_Version='Mime-Version: 1.0'
|and table2. 'Date'='Date: Mon, 9 Apr 2001'
|and table2.Message_ID='Message-ID:<123>')

In Query 1, there are six filters in each group. Only two conditions are exactly same (table1.Mime_Version='Mime-Version:1.0' and table1.User_Name=table2.User_Name) in both the groups as only these conditions will be pushed by the existing optimizer. These are thus the strict filters and the remaining are loose filters. These loose filters are taken care of by our proposed approach. Table 1 shows the difference between the performance of query execution by the existing and proposed approaches using different sizes of data for each comparative execution.

Table 1. Execution of Query 1.

S. no.	Input (table1 & table2)	Existing approach		Proposed approach	
		Duration of execution	Shuffle memory	Duration of execution	Shuffle memory
1.	159.4 MB & 954.0 MB	4 min	62.7 MB	33 s	47.8 KB
2.	1124.4 MB & 5.3 GB	53 min	95.7 MB	3 min	64.0 KB

Query 2:

```
Select count (*) FROM table1 join table2
|where (table1.User_Name=table2.User_Name
|and table1.Subject='Subject: Harper Deals'
|and table1.Mime_Version='Mime-Version: 1.0'
|and table2. 'Date'='Date: Tue, 3 Apr 2001') | or
|(table1.User_Name=table2.User_Name
|and table1.Subject="Subject: Tony's deals"
|and table1.Mime_Version='Mime-Version: 1.0'
|and table2.'Date'='Date: Mon, 9 Apr 2001'
|and table2.Message_ID='Message-ID: <12321401.1075840995900>')
```

In Query 2, only one condition (table1.Mime_Version='Mime-Version: 1.0') is pushed by the existing method before the join operation. The remaining filters are untouched in both groups. These filters will be pushed by the proposed algorithm. table1.Subject='Subject: Harper Deals' and table1.Subject="Subject: Tony's deals" will both be pushed towards table1 and will be evaluated before joining. table2.'Date'='Date: Tue, 3 Apr 2001 ', table2.'Date'='Date: Mon, 9 Apr 2001', and table2.Message_ID='Message-ID: <12321401.1075840995900>' will all be pushed towards table2 and will be evaluated before joining. The result from the above two tables will be used for join operation.

Table 2 shows the difference between the memory and time required during execution of Query 2 by the existing and proposed approaches. The proposed approach gives better results compared to the existing approach.

Table 2. Execution of Query 2.

S. no.	Input (table1 & table2)	Existing approach		Proposed approach	
		Duration of execution	Shuffle memory	Duration of execution	Shuffle memory
1.	159.4 MB & 954.0 MB	4.1 min	61.1 MB	15 sec	43.5 KB
2.	1124.4 MB & 5.3 GB	1.1 hr	92.7 MB	3 min	57.6 KB

Query 3:

```
Select count (*) FROM one1 as table1 join one2 as table2
|where (table1.User_Name=table2.User_Name
|and table1.Subject='Subject: Harper Deals'
|and table1.To='To: andrew.lewis@enron.com'
|and table2. 'Date'='Date: Tue, 20 Mar 2001'
|and table2.File_No='15.') | or
|(table1.User_Name=table2.User_Name
|and table1.Subject="Subject: Tony's deals"
|and table2. 'Date'='Date: Mon, 9 Apr 2001'
|and table2.File_No='15.'
|and table2.Message_ID='Message-ID: <12321>')
```

In Query 3, only one condition (table2.File_No='15.) is the same in both groups, so it will be pushed before joining.

Filters table1.Subject='Subject: Harper Deals', table1.To ='To: andrew.lewis@enron.com', and table1.Subject="Subject: Tony's deals"2001' will be pushed towards table1 and thus the data of table1 will be refined.

Similarly, filters table2.'Date'='Date: Tue, 20 Mar 2001', table2.'Date'='Date: Mon, 9 Apr 2001', table2.File_No='15.', and table2.Message_ID='Message-ID: <12321> will be evaluated by table2.

The resultant data from both tables will be considered for the join operation. Table 3 shows the difference between the memory and time required during the execution of Query 3 by the existing and proposed approaches. The memory consumption is much less with the proposed approach compared to the existing approach while the duration of execution does not change considerably.

Table 3. Execution of Query 3.

S. no.	Input (table1 & table2)	Existing approach		Proposed approach	
		Duration of execution	Shuffle memory	Duration of execution	Shuffle memory
1.	159.4 MB & 954.0 MB	50 s	19.1 MB	42 s	924.0 B
2.	1124.4 MB & 5.3 GB	3.3 min	32.7 MB	3.2 min	1341.0 B

Query 4:

Select count (*) FROM table1 join table2
|where (table1.User_Name=table2.User_Name
|and table1.Subject='Subject: Harper Deals'
|and table1.To='To: andrew.lewis@enron.com'
|and table2. 'Date'='Date: Tue, 20 Mar 2001'
|and table2.File_No='15.')

Query 4 is a scenario in which only one group of filters exists, so all the filters will be pushed by the existing system. Thus, all filters are strict filters in this query and the proposed approach does not have to look for any other filters because there is no loose filter. Subsequently, both the existing and the proposed approaches will use the same amount of shuffle memory and time.

Table 4 represents that there is no difference between the execution performance of the existing and proposed algorithms for Query 4 because there is no loose filter in the query.

Table 4. Execution of Query 4.

S. no.	Input (table1 & table2)	Existing approach		Proposed approach	
		Duration of execution	Shuffle memory	Duration of execution	Shuffle memory
1.	159.4 MB & 954.0 MB	64 s	158.0 B	64 s	158.0 B
2.	1124.4 MB & 5.3 GB	3.2 min	163.0 B	3.2 min	163.0 B

Query 5:

Select count (*) FROM join table2
|where (table1.User_Name=table2.User_Name
|and table1.Subject='Subject: Harper Deals'
|and table1.To='to: andrew.lewis@enron.com'
|and table2. 'Date'='Date: Tue, 20 Mar 2001'
|and table2.File_No='15.') | or
|(table1.User_Name=table2.User_Name
|and table1.Subject="Subject: Tony's deals"
|and table2. 'Date'='Date: Mon, 9 Apr 2001'
|and table2.Message_ID='Message-ID: <12321 >')

In Query 5, no filter is the same in both groups of filters. Therefore, there is no strict filter and all the filters are loose filters. The existing approach will not push any filter towards any table of join operation. By the proposed approach, table1.Subject='Subject: Harper Deals', table1.To='to: andrew.lewis@enron.com', and table1.Subject="Subject: Tony's deals" will all be pushed towards table1 to get filtered.

table2.'Date'='Date: Tue, 20 Mar 2001', table2.File_No='15.', table2.'Date'='Date: Mon, 9 Apr 2001', and table2.Message_ID='Message-ID: <12321 >' will all be pushed to table2 and evaluated on table2. The above resultant data will be fed in the join operation.

Table 5 is proof of efficient utilization by the proposed method during shuffle memory because there is a huge difference between the time and memory required by both the existing and previous methods for executing Query 5. It shows that the introduced method gives a way to utilize memory efficiently. After executing all the queries by the existing and proposed approaches, it is concluded that the execution time and the memory consumption during the shuffling process are much lower by the proposed approach compared to the existing approach.

Table 5. Execution of Query 5.

S. no.	Input (table1 & table2)	Existing approach		Proposed approach	
		Duration of execution	Shuffle memory	Duration of execution	Shuffle memory
1.	159.4 MB & 954.0 MB	3.5 min	78.3 MB	40 s	84.7 KB
2.	1124.4 MB & 5.3 GB	1.4 h	124 MB	3.2 min	107.5 KB

Figure 5 represents the time required by executing all the queries by the existing and the proposed approach, where table1 has 159.4 MB amount of data and table2 has 954.0 MB amount of data. There is a significant change in both time and memory consumption between the existing and the proposed approach of execution.

Figure 6 represents the time required for executing all the queries by the existing and the proposed approach, where table1 has 1124.4 MB amount of data and table2 has 5.3 GB amount of data.

Figure 7 represents the memory required during the shuffling process for executing all the queries by both the approaches. Here table1 is of size 159.4 MB and table2 is of size 954.0 MB.

Figure 8 represents the memory required during the shuffling process during execution of all the queries by both the approaches, where table1 is of size 1124.4 MB and table2 is of size 5.3 GB.

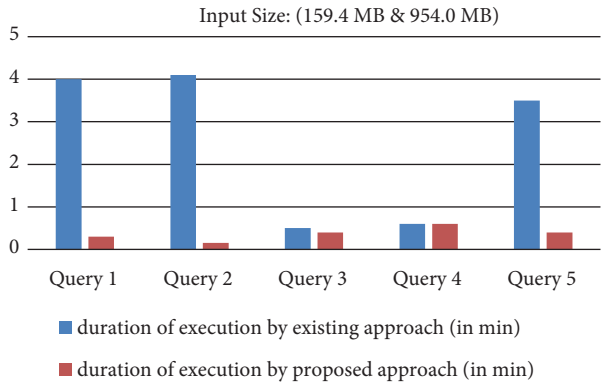

Figure 5. Time required executing all the queries (159.4 MB & 954 MB).

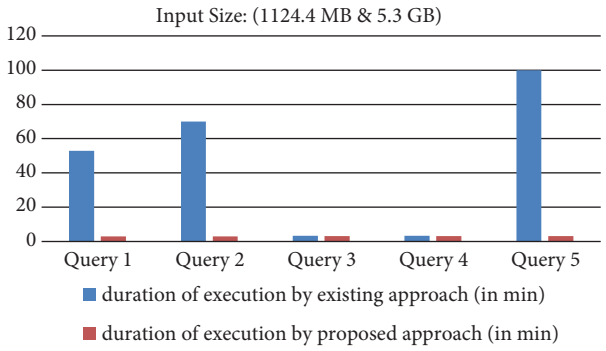

Figure 6. Time required executing all the queries (1124.4 MB & 5.3 GB).

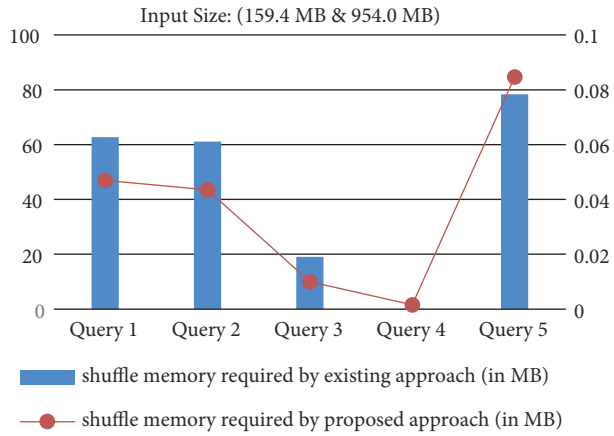

Figure 7. Memory required executing all the queries (159.4 MB & 954 MB).

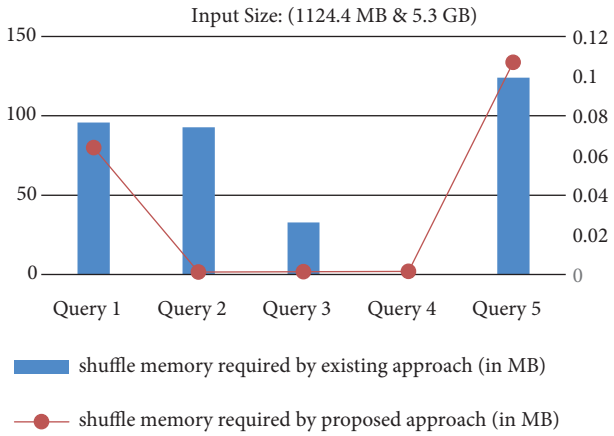

Figure 8. Memory required executing all the queries (1124.4 MB & 5.3 GB).

5. Conclusion

The proposed approach reduces the data before joining, which leads to higher performance of a job. It results in a faster query processing without loading irrelevant data, which is not possible by the existing approach. Therefore, much lower amounts of memory and time are required during the shuffling process compared to the existing approach.

If there are no loose filters in the query, then both the existing and the proposed solutions take the same amount of time and memory.

6. Future work

Discussion about the strength of the proposed approach has been done above, but still, to improve this solution, new ideas can be mixed into this approach. This introduced approach fits specific queries and in that case our proposed algorithm will improve the memory usage and execution time.

The next target will be to optimize the queries of different formats up to a significant level in the future so that a user does not need to think about the semantics of the query or manual optimizations. All the possible optimizations will be done automatically.

References

[1] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. HotCloud 2010; 10: 95.

[2] Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A et al. Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM International Conference on Management of Data; 2015. pp. 1383-1394.

[3] Gopalani S, Arora R. Comparing Apache Spark and Map Reduce with performance analysis using k-means. International Journal of Computer Applications 2015; 1: 113.

[4] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation; 25 April 2012. p. 2.

[5] Rana N, Deshmukh S. Shuffle performance in Apache Spark. International Journal of Engineering Research and Technology 2015; 4: 177-180.

[6] Rana N, Deshmukh S. Performance improvement in Apache Spark through shuffling. International Journal of Science, Engineering and Technology Research 2015; 4: 1636-1638.