Scalable Query Optimization for Efficient Data Processing using MapReduce

Yi Shan #1, Yi Chen 2

School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ, USA
School of Management, New Jersey Institute of Technology, Newark, NJ, USA

1 yshan1@asu.edu 2 yi.chen@njit.edu

***Abstract*—MapReduce is widely acknowledged by both industry and academia as an effective programming model for query processing on big data. It is crucial to design an optimizer which finds the most efficient way to execute an SQL query using MapReduce. However, existing work in parallel query processing either falls short of optimizing an SQL query using MapReduce or the time complexity of the optimizer it uses is exponential. Also, industry solutions such as HIVE, and YSmart do not optimize the join sequence of an SQL query and cannot guarantee an optimal execution plan. In this paper, we propose a scalable optimizer for SQL queries using MapReduce, named SOSQL. Experiments performed on Google Cloud Platform confirmed the scalability and efficiency of SOSQL over existing work.**

I. Introduction

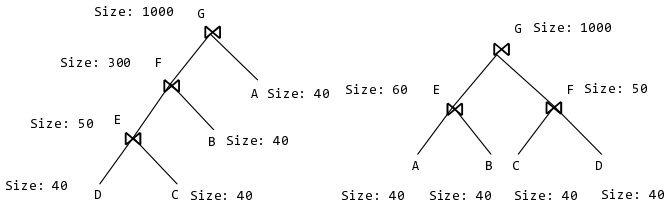
It becomes increasingly important to process SQL queries on big data. Query processing consists of two parts: query optimization and query execution. The query optimizer first generates a query plan. Each node in the query plan encapsulates a single operation that is required to execute the query. Then based on the query plan, the query optimizer generates an execution plan, defined as an ordering of the nodes in the query plan. An execution plan stands for an execution sequence of all operations. Finally, the query engine will follow the query execution plan to execute the query. Different algorithms have been proposed to efficiently implement each operation, such as selection, join, etc.

There has been decade long research on processing SQL queries in parallel, as summarized in a survey book [12]. For an input SQL query, a query optimizer first generates the optimized query plan. It has been shown that a query plan in the structure of either a deep left tree (DLT) or a bushy tree (BT) is typically effective, which can be efficiently generated. Then the query optimizer constructs an optimal execution plan by breaking the optimal query plan into a sequence of binary join operations. In parallel systems, a multi-way join must be broken down to binary joins to execute, since each parallel job is able to execute only one binary join. This is because existing join algorithms are designed for binary joins. Finally, each binary join operation is carried out by a parallel job whose output might be used as input to other jobs [7].

Example 1.1: Let us look at a simple SQL query Q_1 with three joins, shown in Figure 1. Two optimized query plans are generated by the existing work [7], [9], [14], a DLT query plan shown in Figure 2(a), and a BT query plan shown in Figure 2(b). For the BT query plan, the query optimizer further generates a query execution plan consisting of three parallel jobs, one for each binary join operation, as illustrated in Figure 3(a). We have three jobs: table A and B are joined with result table E, then table C and D are joined with result table F, and then, the result table E is joined with table F. ■

```
select *
from A, B, C, D
where A.JK1 = B.JK1 and B.JK2 = C.JK2 and C.JK3 = D.JK3
```

Fig. 1. Query Q_1



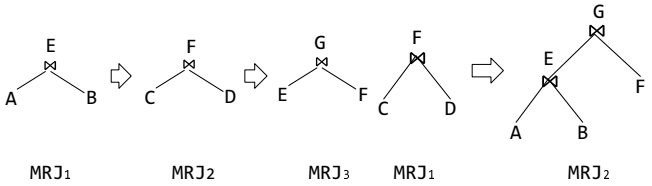
(a) QP_1 (b) QP_2

Fig. 2. Query Plans of Query Q_1

In recent years, MapReduce has been widely acknowledged as an effective programming model to process big data. MapReduce follows a shared-nothing model and leverage commodity hardware to effectively handle large-scale data. Studies have been performed on how to leverage MapReduce framework for efficient SQL query processing on big data. Many studies focus on how to implement join operations in MapReduce framework, in particular, using one MapReduce Job, denoted as *MRJ*, to execute a single join operation [5], [13], [16], [4], [8], [10]. These studies provide great insights into how to run a MRJ. However, limited investigation has been done about how to find the best way to break a query into a set of MRJs. In other words, limited research has been studied on query optimization for SQL queries on MapReduce framework.

One option is to leverage the query optimizers developed for parallel SQL query processing to generate a query plan and an execution plan when processing SQL queries on MapReduce framework [15]. However, it is observed that in MapReduce framework multi-way joins are not necessarily broken down into binary joins to execute, and a MRJ can execute a multi-way join [4].

Example 1.2: Continuing our example, for query plan in Figure 2(b), the execution plan shown in Figure 3(a) is a valid execution plan for Q_1 in MapReduce framework. Another



(a) EP_{21} (b) EP_{22}

Fig. 3. Execution Plans of QP_2

execution plan is shown Figure 3(b) using MapReduce, where table C and D are joined with result table F by MRJ_1. Then table A, B and F are joined by MRJ_2 as a multi-way join. ■

Clearly not all execution plans are equal in terms of efficiency. [4] shows that sometimes having one MRJ for multi-way join can be more efficient than using a set of binary joins due to data movement cost . Thus the traditional approach used in parallel SQL query processing, which constructs an execution plan by breaking the query plan into a sequence of binary join operations, may not be optimal for processing SQL queries in MapReduce framework.

In this paper, we studies the problem of how to optimize query execution plan for executing SQL queries in MapReduce framework with the goal of improving the overall query processing efficiency. The generated query execution plan needs to be effective to achieve efficient query execution, and its generation needs to be efficient. As our contribution, we developed a prototype system, SOSQL, Scalable Optimizer of SQL, that takes an SQL query as input and generates a query execution plan that consists of a sequence of MRJ, where an MRJ can be a binary join or a a multi-way join. This problem is challenging because finding the optimal execution plan is NP-hard. To address this challenge, we identify a chain join property of executing join operations on MapReduce, and prove that each join operation executed on MapReduce must be a single branch tree structured query plan. Then we propose a polynomial algorithm to optimize the query execution plan. The proposed solution for query execution plan optimization has a time complexity of $O(n^2)$, where n is the number of input tables of the input SQL query. We have performed empirical studies on Google Cloud Platform using TPC-H datasets. The evaluation shows that our approach has significant speedup over HIVE 1.1.0, which represents a common industry solution. SOSQL also largely outperforms existing work that uses exponential algorithms to optimize execution plan, such as AQUA [15]. The average speedup of SOSQL over HIVE is 1.64, while 1.47 for AQUA.

II. Related Work

There are a lot of studies on parallel processing for SQL queries. [7], [9], [14] proposes a general framework to execute an SQL query. The query engine first generates a query plan. Then an execution plan is generated by breaking the query plan into a sequence of binary join operations. Finally, each binary join operation is carried out by a job. An survey book [12] provides a good overview of this field.

There are a lot of studies on leveraging MapReduce framework for SQL query processing. The focus of the studies is join algorithm implementation in MapReduce framework. [5], [13], [16], [4], [8], [10], [11] studies the use of one MRJ to execute a single join operation such as equi-join and theta-join. [5] proposes four binary equi-join algorithms using MapReduce, among which repartition join is the most widely used one. [13] proposes a workload partition algorithm to handle binary theta-join algorithms using MapReduce. [4] extends the above two and proposes a multi-way binary join algorithm. [16] proposes a multi-way theta-join algorithm using MapReduce.

Limited studies have been performed on query optimization for processing SQL queries on MapReduce framework, in particular, how to find the best way to break an SQL query into multiple MRJs to achieve high efficiency. [15], [16] discusses how to break an SQL query into multiple multi-way joins, using an algorithm of exponential time complexity. HIVE, YSmart, PIG and other industry solutions for SQL query processing using MapReduce do not optimize the join sequences.

III. Problem Statement

Each SQL query can be represented by a join graph. Each vertex stands for a table and each edge stands for a join between two tables of the query. Our optimization goal is to find the optimal way to break the join graph into a set of sub-graphs and assign each sub-graph to an MRJ such that all edges of the join graph are executed with minimal time. We name a sequence of MRJs as an MRJ assignment (MRJA). Furthermore, we use $Cost(MRJ)$ to indicate the cost of an MRJ and $Cost(MRJA)$ for an MRJA. Then, formally we have the problem statement and optimization goal of query processing using MapReduce shown below:

Definition 3.1:[Problem Statement]

Given an SQL query Q, find an MRJA such that each of its MRJs executes a sub-query of Q and combined return the results of Q s.t. $Cost(MRJA)$ is minimized. ■

For example, Figure 3 shows 2 possible MRJAs to run Q_1. Our optimization goal is to find the optimal one among all possible MRJAs of Q_1.

To define the cost model, we first discuss the join algorithms used. There are four types of join algorithms used in MapReduce framework: repartition join, directed join, broadcast join and semi-join. Among these four join algorithms, repartition join is the most widely used [5], [4]. [4] proposes replicate join that extends repartition join to support multi-way join and improve the efficiency of query processing. We adopt this join algorithm of a single MRJ in this paper. As shown in [6], to join n tables $T_1, T_2, ...T_n$, each replicate join could be represented in simple relational algebra as $T_1 \bowtie T_2 \bowtie ...T_n$, which is a chain join. A chain join means input tables are joined one after another in sequential order.

We define the cost of an MRJA as the summation of the cost of its MRJs. Formally, for an MRJA with a sequence of q MRJs s.t. $MRJA = (MRJ_1, MRJ_2, ..., MRJ_q)$, we have $Cost(MRJA) = \sum_i Cost(MRJ_i)$. . For $Cost(MRJ_i)$, there is much research defining the cost model of replicate join using an MRJ [16], [4], [15], [13]. We take a simple cost

model which considers I/O cost dominates MRJ run time, as in [5]. That is, the cost model of each MRJ is measured as the sum of total mapper input size and shuffle data size.

Continuing our example, let us discuss how to compute $Cost(MRJ_2)$ with 4 reducers. We have the sum of total input $= |A| + |B| + |F| = 130$. To compute shuffle data size, we first need to calculate for how many times each tuple, tu, is replicated from mappers to reducers. Suppose there are r reducers, and m join keys among all tables. If tu has t of the m join keys, then tu will be replicated for $r^{\frac{(m-t)}{m}}$ times. $Cost(MRJ_2)$ has 2 join keys, $JK1$ and $JK2$. So each tuple in table A and table F are replicated twice, $4^{\frac{(2-1)}{2}}$. For each tuple $tu \in B$, it is replicated for once. So the shuffle data size $= 2 \times |A| + |B| + 2 \times |F| = 220$. So we have $Cost(MRJ_2) = 130 + 220 = 350$. Similarly, for MRJ_1 in Figure 3(b), we have $Cost(MRJ_1) = 160$. As a result, for the corresponding MRJA, we have $Cost(MRJA) = 510$.

IV. Query Optimization

Our SQL query processing system, SOSQL, consists of three major components. First, the query plan optimizer generates the optimal query plan for an SQL query. Then, the execution plan optimizer takes the optimal query plan together with MapReduce framework settings as input and generates the optimized execution plan and its corresponding MRJA. Finally, we run MRJs of the MRJA using an MapReduce query engine.

A. Query Plan Optimizer

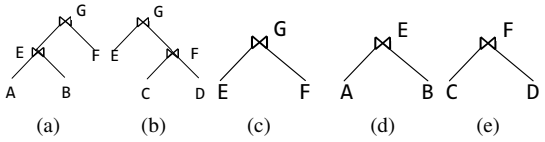
As proposed in [9], an optimal query plan stands for the conceptually optimal join sequence in a tree structure.

For example, Figure 2 shows two different query plans of Q_1 in Figure 1. For Q_1, query plan QP_1 in Figure 2 is a DLT query plan, while QP_2 is a BT query plan. In QP_2, join operation G is dependent on E and F, however E and F are independent on each other. In QP_1, join operation G is dependent on F which is dependent on E. For QP_1, it can be represented as $D \bowtie C \bowtie B \bowtie A$ in relational algebra, which is a chain join. For QP_2, it can be represented as $(A \bowtie B) \bowtie (C \bowtie D)$, which is not a chain join. With the estimated size of intermediate tables annotated on the graph, we have $Cost(QP_1) = |D|+|C|+|B|+|A|+|E|+|F|+|G| = 1,510$ and $Cost(QP_2) = 1,270$.

Given all table sizes, we refer to [9] which proposed polynomial algorithms with time complexity $O(n^2)$ to find the optimal DLT and BT structured query plans.

B. Query Execution Plan Optimizer

With an optimized query plan generated, in this section we discuss how to optimize the execution plan and the corresponding MRJA. First we will define optimal query execution plan based on optimal query plan. Then, for both DLT and BT structured query plans, we devise algorithms to optimize the query execution plan respectively.



(a) (b) (c) (d) (e)

Fig. 4. Query Optimization of QP_2

1) Properties of Query Execution Plan: As introduced in Section III, replicate join is a chain join in relational algebra. Also, we see from the above example that a DLT query plan can be represented as a chain join in relational algebra as well. We prove that each DLT query plan can be implemented as an replicate join, which a property that we use to guide the generation of query execution plan. This indicates that each MRJ of a replicate join must execute a DLT query plan, such as Figure 3. Next, we leverage this theorem to devise efficient query execution plan generation optimizer.

For example, one query execution plan for query plan QP_2 in Figure 2(b) would be using a single MRJ to execute it. However, based on the property above, each MRJ must execute a DLT query plan. Because QP_2 is not a DLT query plan, it is infeasible to use one MRJ to execute QP_2. On the other hand, the $MRJA = (MRJ_1, MRJ_2)$ in Figure 3(b) is a valid execution plan.

From the above discussion we know that given a query plan QP, a query execution plan is an MRJA with a sequence of q MRJs such that each MRJ_i is a DLT structured sub-tree of QP. Also MRJ_i must observe no dependency when being executed. Then the optimal query execution plan is the query execution plan with the minimum cost.

Figure 3(a) and Figure 3(b) show two query execution plans for the BT structured query plan QP_2 in Figure 2(b). For DLT structured query plan QP_1 in Figure 2(a), we can generate similar execution plans. Next we will discuss how to generate the optimal query execution plan for input query plans.

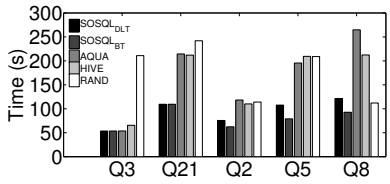
2) Dynamic Programming Search for Optimal Execution Plan: As an intuitive algorithm, using exhaustive search to find the optimal query execution plan for BT query plan results in exponential time complexity. To boost performance, we propose a dynamic programming algorithm with $O(n^2)$ time complexity.

For example, for QP_2 with root G, we want to find the MRJA of the optimal execution plan with the minimum cost.

Step 1. Based on the property of Section IV-B1, we know that the root of a query plan is executed in a DLT sub-plan by an MRJ. So we enumerate all DLT sub-plans, Figure 4(a) to Figure 4(c), in which G is executed in and identify the one inducing the minimum cost. The cost of executing each DLT sub-plan alone is 350, 380 and 220.

Step 2. After removing all edges and input tables of the DLT sub-plan chosen in Step 1, the original query plan becomes a forest of BT sub-plans. If we remove the DLT sub-plan in Figure 4(a), Figure 4(e) is left. If we remove the DLT sub-plan in Figure 4(b), Figure 4(d) is left. If we remove Figure 4(c), Figure 4(d) and Figure 4(e) are left.

Step 3. To calculate the total cost of the DLT sub-plan in Step 1, we first need to calculate the cost of the BT sub-plans in Step 2 because they must be executed beforehand. The cost



(a) 100GB

Fig. 5. Overall Query Processing Time

of executing Figure 4(d) or Figure 4(e) is 160 as shown in Section III.

Step 4. For each DLT sub-plan of Step 1 and its corresponding BT sub-plans in Step 2, they form a MRJA. So the cost associated with the DLT sub-plan in Figure 4(a) is the sum of executing Figure 4(a) and Figure 4(e), 510. Similarly, the cost associated with Figure 4(b) and Figure 4(c) are 540 and 540. Then, we return the MRJA with the minimum cost from Step 4, which is Figure 3(b).

Based on the above example, we can see a clear optimal structure. By enumerating the DLT sub-plan in which the root of a query plan is executed, we can break down the original BT query plan into a set of smaller BT sub-plans. Furthermore, to optimize the original BT query plan, we must optimize each generated BT sub-plan.

Given a BT query plan QP, we use $DLT_l(v, QP)$ to indicate the DLT subtree of QP rooted at node v with height l. For example, $DLT_2(G, QP_2)$ indicates both Figure 4(a) and Figure 4(b), while $DLT_1(G, QP_2)$ indicates Figure 4(c).

Furthermore, we use $dp_{QP}[v]$ to indicate the cost of the optimal execution plan of the sub-plan rooted at v of QP. We use $LN(.)$ the indicate the leaf node set of a tree. For example, $dp_{QP_2}[G]$ indicates the cost of the optimal execution plan of QP_2. Then for a query plan QP with height h, we have the optimal function in Equation 1.

$$dp_{QP}[v] = Min_{1 \le l \le h}(Cost(MRJ(DLT_l(v, QP))) + \sum_{v' \in LN(DLT_l(v,QP)} dp[v^{'}])), \quad (1)$$

where $MRJ(DLT_k(v, QP))$ is an MRJ executing $DLT_l(v, QP)$. $v^{'}$ $dp[v^{'}] = 0$ if $v^{'}$ is an input table because it does not need to be executed.

V. Experiments

Our experiments run on a 32-node Hadoop 1.2.1 cluster deployed on Google Cloud Platform (GCP) [2] with 64 cores, 240GB memory and unlimited storage using default configuration. 64 reducers are created for each MRJ. We generate optimized query plan and execution plan on a single machine with 3.1GHz Intel Core i5-2400 and 16GB memory running Windows 7. We performed experiments on the standard TPC-H dataset with 100GB data size [3]. We present the performance analysis of processing Q3, Q21, Q2, Q5 and Q8. These queries represent queries with varying sizes, with the number of tables in a query to be 3, 4, 5, 6 and 8, respectively. The number of input tables in TP-C queries vary from 3 to 8.

We compare SOSQL with three systesm. AQUA [15] represents existing work for processing SQL queries in MapReduce that uses exponential algorithms for execution plan optimization. HIVE 1.1.0 [1] represents common industry solutions, where one MRJ is used for each binary join at a time, which is also the strategy used in existing work in parallel query processing. Furthermore, we compare our system with a baseline system which randomly chooses an MRJA to execute an SQL query, named RAND. For the proposed SOSQL system, since we generate query execution plans based on both DLT and BT query plans, we test both approaches, denoted as $SOSQL_{DLT}$ and $SOSQL_{BT}$, respectively.

We compare the overall query processing time of SOSQL with all the other three systems on TPC-H dataset of size 100GB, as shown in Figure 5. Query processing time includes both query optimization time and query execution time. With the same data size and query, different systems have different query processing times. As shown in Figure 5, $SOSQL$ always achieves the best performance with over 130% improvement over other systems. Also, we notice that $SOSQL_{BT}$ performs slightly better than $SOSQL_{DLT}$, which is consistent with the observation made in the existing work [15].

VI. Conclusions

In this paper, we proposed SOSQL which is a scalable SQL query optimizer using MapReduce. Based on our extensive evaluation of SOSQL using standard TPC-H dataset and Google Cloud Platform, we conclude that SOSQL improves the efficiency of SQL query processing in terms of both query optimization and query execution.

VII. Acknowledgements

This work is partially supported by NSF CAREER Award IIS-0845647, Google Cloud Service and the Leir Charitable Foundations.

References

[1] http://hive.apache.org/.
[2] https://cloud.google.com.
[3] http://www.tpc.org/tpch/.
[4] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. In *IEEE Trans. Knowl. Data Eng.*, 2011.
[5] S. Blanas, J. M. Patel, and V. E. and. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD Conference*, 2010.
[6] J. Chandar. Join algorithms using map/reduce. Master's thesis, University of Edinburgh, 2010.
[7] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *ICDE*, pages 58–67, 1992.
[8] C. Doulkeridis and K. Nrvg. A survey of large-scale analytical query processing in mapreduce. In *VLDB J.*, pages 355–380, 2014.
[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD 1992*, pages 9–18, NY, USA, 1992.
[10] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. In *SIGMOD Record*, 2011.
[11] X. Lin, Y. Ye, and S. Ma. Mrpacker: An sql to mapreduce optimizer. In *CIKM*, CIKM '13, 2013.
[12] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.
[13] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD Conference*, pages 949–960, 2011.
[14] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. In *ACM Trans. Database Syst.*, 1984.
[15] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, page 12, 2011.
[16] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. In *PVLDB*, 2012.