JOMR: Multi-join optimizer technique to enhance map-reduce job

Dr. Samah A. Senbel, Sacred Heart University
JOMR: Multi-join Optimizer Technique to Enhance Map-Reduce Job

Mina Samir Shenouda
Dept. of Information Systems
Arab Academy for Science, Technology & Maritime Transport
Cairo, Egypt
mena_samir@hotmail.com

Mohamed Helmy Khafagy
Dept. of Computer Science
Fayoum University
Fayoum, Egypt
Mhk00@fayoum.edu.eg

Samah Ahmed Senbel
Dept. of Computer Science
Arab Academy for Science, Technology & Maritime Transport
Cairo, Egypt
senbel@aast.edu

Abstract— Map-Reduce is a programming model and execution an environment developed by Google to process very large amounts of data. Query optimizer is needed to find more efficient plans for declarative SQL query. In classic databases, join algorithms are optimized to execute the entire query result, but they ignore the importance of tables order especially in multi-join query. But we can see that the orders for tables are an important factor to get the best performance of a query plan and will be very effective in performance when join tables content huge number of rows in addition to more than one join operation. In this paper we proposed a new technique called JOMR (Join Order In Map-Reduce) that optimizes and enhances Map-Reduce job. This technique uses enhanced parallel Travel Salesman Problem (TSP) using Map-Reduce for improving the performance of query plans according to change the order for join tables. Also we build a cost model that supports our algorithm to find best join order. We will focus on Hive especially multi-join query and our experiments result for JOMR algorithm proving the effectiveness of our query optimizer and this performance is improved more when increasing the number of join and size of data.

Key words —Query Optimization, Hadoop, Map-Reduce, Hive, Column Statistics, Join.

I. INTRODUCTION

A. Introduction to Map-Reduce

Map-Reduce[1] is a programming model Execution environment, developed on Google to process large amounts of information or the order of Gigabytes, Terabytes, Peta-bytes or Zetta-bytes, parallel and distributed on a cluster composed of nodes utility. Map-Reduce is used by many companies to carry out tasks Business Intelligence, as market trends, introduction of a new product, data mining, and so on. Also this technique featured with availability using Replication [2] and has load balance [3] to ensure the best use of resources. Massively Parallel Processing (MPP) [4] data warehouse systems such as Netezza, Teradata, AsterData, Greenplum, Oracle, among others, provide services and products (hardware and software) Data warehouse to process and analyze large amounts of data with time and detail suitable. To run any applications on Hadoop Map-Reduce, first must transform and code this application as a map and reduce functions. In a relational database management system (RDBMS) mostly using SQL language but in Map-Reduce using Hive. Hive is an opensource software build on Hadoop Map-Reduce can analyze huge volumes of data. Hive supports and query functionality to run project on this data similar to functionalities using a relational database management system (RDBMS).Hive’s task is transforming the query into a Map-Reduce jobs. Hive can be used to generate tables and store data into tables using the Hadoop File System (HDFS) or other Distributed File system [5,6,7], which then can be accessed it data using a Hive query language called HiveQL. When use HiveQL to extract data bases of two phases. The first phase is parsed and translated into a query execution plan, the second took this logical plan according to the first phase then optimized using query optimization in hive components and then moved into maps and reduce phases.

B. Query Optimization in Map-Reduce

HadoopDB[8] is a hybrid system, aiming to use the best features of Map-Reduce and parallel DBMSs. The basic idea of HadoopDB is to put copies of a DBMS at each nodes of a grid and connect with these nodes by Hadoop to parcel work among them as the task coordinator and network communication. Major benefits when compared with massively parallel DBMS. Query processing on each node is improved by assigning as much work as possible to the local database. By this way we can collect benefits of query optimization by local DBMS. As has been well recognized in conventional query processing, good plans can indeed improve query performance by orders of magnitude [9]. In current systems, such as Pig [10] and Hive [11, 12], users submit their queries in the corresponding query language supported by the system. The query designation to a sizable voluminous degree fine-tunes the concrete query plans utilized by the underlying system to evaluate the queries. Therefore, as in traditional DBMS generates the best query plan by using a query optimizer. The problems of finding the optimal join ordering executing a query is a combinatorial optimization problem. Each query can be executed in many query plans and each plans generated the same result but the different in the cost of processing each query plan. The response time of a query is a parameter type to evaluate each plan. To minimize the cost of processing time of a query

Copyright© 2014 by Faculty of Computers and Information–Cairo University

PDC-80
execution and increase the performance is by using some statistics on columns that describe the data stored in the database. By column statistics can be used to determine the join order for many tables or to estimate cost of join between two tables by evaluating number of values that are possibly generating in the output.

C. Join algorithms

In the following section we will describe different types of traditional join algorithms and what differences between join algorithms in database systems and join algorithms in Hadoop.

1) Classic join algorithm

Before jumping on joining algorithms using Hadoop review the currently exiting join algorithms in standard DBMS.

a) Nested Loops Join

It is one of very simple join algorithms. It allows joining two tables based on any join condition. This join algorithm can only perform equi-joins.

b) Sort-Merge Join

The basic idea of the sort merge join algorithm, given two tables T1 and T2, the join strategy first sorts both tables on the join condition and then merges the two tables. The sorting phase is grouping all tuples with the same value in the join column together then it is easy to partitions with the same value. This algorithm is avoiding listing or the cross product between two tables.

c) Hash join

The hash join algorithm consists of two phases. The first phase is building the smaller table that is loaded in memory hash table. The second phase is to inquire; the largest table is scanned and joined with the same values in the hash table.

2) Join algorithm in Hadoop

We will describe different types of join algorithms in Map-Reduce [13].

a) Repartition join

Repartition join is most general reduce-side join in the Map-Reduce. In this join algorithm in which E and D are dynamically partitioned on the join key and the pairs of the corresponding with key are joined. These join algorithms like sort-merge join in the parallel. The repartition join is works on only one Map-Reduce job. Each mapper tags a split of two relations to find each row come from which table. The map task contents the rows with this product table, the output taken away each key with the row as a (key,value). Then the outputs are separated, merged and sorted by the Hadoop. All the rows for each join key are grouped together and finally to supply a reducer.

b) Broadcast join

Broadcast join algorithm applicable when the size of table D is smaller than another table E, i.e. |D| ≪ |E|. As in repartition join must moving both tables D and E across the network, this join algorithm move only the smaller table on each node. By this strategy avoids I/Os for sorting and moving on both tables and avoid network loaded by move the largest table E. The broadcast join is working only on one Map-Reduce job. On each node, content the smaller table D is retrieved from the D. The map task is used in memory hash tables to find matches via table lookup with values from the other input relation.

c) Semi-join

Semi-join algorithm depends on avoiding sending rows of table D over the network that will not join with table E. Semi-join algorithm implemented in three phases, each separates Map-Reduce job. The first phase in the map task detects the values of unique join keys in partition of table E in memory hash table. The reduce task produces each unique join key, merge all the unique join keys into one file which is not large to fit in memory. The second phase is typical what happened in broadcast join takes the output from phase one into memory hash table then repeat in each rows in table D and output it if join key found. This phase generates list of files Di, one for each partition of table D. The third phase takes the output of phase two to join with table E by using broadcast join algorithm.

d) Per-Split Semi-Join

The per-split semi-join is designed to solve the problem in semi-join. Per-split Semi-join algorithm implemented in three phases, each separate Map-Reduce job. The first phase produces the list of unique join keys in split Ei of table E and stores them in the DFS file. In the second phase, the map task is generating all rows from split of table D in memory hash table, and then reads the unique keys from list of unique join keys in split Ei of table E and inquires for matching values in table D. Finally matched values are outputted. In the third phase the output from phase two are joined with split of table L using directed join.

D. Left deep vs. bushy plan

When the query plan is very large for complex SQL query, most DBMS is used left-deep plan in query optimization [14]. Many real applications use this strategy. However, it may leads to lowly plan for Map-Reduce based query processing. This is because a Map-Reduce job needs to fulfill the internal results of sub join query.

Now, shows two plans for processing in DBMS, the first plan left deep as shown in Fig. 1 is the better because it is an easy pipeline processing as at least one data source is the base table. First produced the result for then pushed the result to join with table T3. Second plan is a pushy plan as shown in Fig. 2 first produce the internal results for both join then pushed the result to last join.
E. Plan Generation Process

- Parser: syntactical/semantically analysis
- Rewriting: optimizations independent of the current database state using parallel TSP in order to join tables.
- Optimizer: optimizations that rely on a cost model and information about the current database state
- The resulting plan is then evaluated by the system’s execution engine.

![Plan generation process diagram](image)

The 9th International Conference on INFOrmatics and Systems (INFOS2014) – 15-17 December
Parallel and Distributed Computing Track

III. THE TRAVELING SALESMAN PROBLEM (TSP)

The Traveling Salesman Problem (TSP) consists of number of cities on a map and the distances between each city, what is the shortest pass to visits each city exactly once and returns to the city it started from. The problem is find shortest path to visit each node [17]. Recently a polynomial time approximation scheme (PTAS) was discovered for the Euclidean TSP problem. However, with Moore’s law nearing the end of its lifetime and parallel and cloud computing gaining occurrence in the past few years, found that efficient parallelization of existing algorithms will earning importance. The algorithm is used to solve problem of TSP and Parallelized over Map-Reduce is Tabu search [18]. Tabu search algorithm is depending on defines all possible moves one of this move can change the pass, two cities in the tour could exchange two edge crossing each other.

Algorithm 1: Tabu search

```
1: Set S = initial()
2: OptimalSol = S
3: tables = n
4: visited = null
5: for i=0 to n do
6:   selectlist = null
7:   for (Selector in Neighborhood)
8:     if (not contains (Selector, visited))
9:       selectlist = selectlist + Selector
10:  end
11: end
12: Selector = LocateBest(selectlist)
```
13: if(fitness(Selector) > fitness(OptimalSol))
14:   visited=Diff( Selector, OptimalSol)
15:   OptimalSol = Selector
16: while(size(visited) > maxvisitedsize)
17:   Finish(visited)
18: end
19: end
20: return(OptimalSol)

This algorithm returns the optimal solution to visit each node one time with minimum cost.

IV. JOMR ALGORITHM

In this algorithm we propose JOMR (Join Order In Map-Reduce) that optimizes and enhances Map-Reduce Job. In Hive, we found that the performance of multi-join query by change join order of tables in multi-join query of a Map-Reduce based system is the cost of saving intermediate results.

A. Cost Model

Cost model is mathematical algorithm used to estimate cost to evaluate between possible plans then can detect which is the best solution. To evaluate the time for query execution plan. We built a cost model for the Map-Reduce framework. The cost model consists of some column statistics to estimate the performance of join query to generate optimal join order for query. In the cost model, we have two types of costs: the first type is I/O costs for example local disk I/O and network I/O and CPU costs. The second type is total cost of processing a Map-Reduce job is used as the metric. Table 1 shows all this parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ri</td>
<td>Cost of read local disk</td>
</tr>
<tr>
<td>Wi</td>
<td>Cost of write local disk</td>
</tr>
<tr>
<td>Rdfs</td>
<td>Cost of read HDFS</td>
</tr>
<tr>
<td>Writefs</td>
<td>Cost of write HDFS</td>
</tr>
<tr>
<td>V</td>
<td>Cost of CPU</td>
</tr>
<tr>
<td>N</td>
<td>Cost of Network I/O</td>
</tr>
</tbody>
</table>

We assume these parameters constant not in our focus. We will focus only about the cost of processing Map-Reduce jobs.

B. Column statistics

Hive supports some statistics at the table level like count of files, raw data size, count of rows, but doesn’t support statistics on column level. To improve the performance of execution query plan collecting some statistics on the data stored in the DB [19]. We will generate various statistics on data collected from column values. Some of this column statistics as following:

- Number of frequency values
  The count of rows in a table may be different from distinct values in this table.
- Number of frequency values after condition

The count of rows in a table after removes the values for the condition.
- Number of Null Values
  Ignore null values from the count of rows when join with another column primary key.
- Height distinct values
  These values describe the count of appearing these values in table, this value refer to the worst case if the query has condition.

By using this statistics we have metadata on the column level gives estimates for accurate time to join query. This query optimization technique has been shown to have huge effect on the execution time for join query in huge table.

C. Optimizer TSP

The first step to optimize the tsp join algorithm is generating matrix content all possible edges and weight between each two edge, must we have number of nodes in our case is tables and determined the possible edges between nodes. To determine the number of edges to nodes we will use equation (1)

\[ E = \frac{n \times (n - 1)}{2} \]  

After calculating the number of edge must known what is the weight for each edge. We used new technique to calculate edge weight. We used column statistics described before. We used all above column statistics by using equation (2) to calculate weight over edge when two tables don’t have any condition in the query. This weight means the cost of join between two tables T1 and T2 when two tables join with others with a Foreign Key (FK)

\[ W = \text{Weight of tables} \]

\[ NC = \text{Number of frequency values after condition} \]

\[ HD = \text{Height distinct values for FK} \]

\[ W_{(T1,T2)} = NC \times \text{Max(HD)} \]  

We used all above column statistics by using equation (3) to calculate weight over edge when two tables have a condition in the query. This weight means the cost of join between two tables T1 and T2 when two tables join with others with a Foreign Key (FK)

\[ N = \text{Number of frequency values} \]

\[ W_{(T1,T2)} = N \times \text{Max(HD)} \]  

We used all above column statistics by using equation (4) to calculate weight over edge when two tables don’t have a condition in the query, don’t have FK with others.

\[ W_{(T1,T2)} = N \times (T1) \times N(T2) \]
D. Topic Identification

Now will be optimized tabu search algorithm to get best join order. In tabu search, the start node must be the last node and in between visit all nodes with minimum cost but in our algorithm change this technique to start from minimum weight of an edge then visit all nodes with minimum weight. After running this algorithm, we will have best join order plan for this query to run with best cost and less operation in multi-join query. The differences between an original query and new join query plan are very high and the cost decreases when the number of the table is increasing.

Algorithm 2 shows pseudo code for JOMR.

Algorithm 2: JOMR
1: Set M = possible_edges()
2: First_edge = Min(M)
3: tables = n
4: visited = null
5: left_selector = left_of_Min(M)
6: Right_selector = Right_of_Min(M)
7: No_of_edges = n*(n-1) / 2
8: Ledges = null
9: Redges = null
10: for i=0 to n do
11: list = null
12: for j=0 to n
13: if(contains (M, left_selector ))
14: if(not contains (Selector, visited))
15: Ledges = Ledges+1
16: end
17: X = Min (Ledges)
19: end
20: if(contains (M, Right_selector ))
21: if(not contains (Selector, visited))
22: Redges = Redges+1
23: end
24: Y = Min (Redges)
26: if( X > Y )
27: list = add ( Y )
28: else
29: list = add ( X )
30: return ( list )

E. Implementation approach

To evaluate this metadata for improving the performance by using column statistics level, we worked in four steps to collect the column statistics level and then to apply the collected statistics to a query in order to optimize the query execution. The first step is parsing input query to detect number of tables in join query and detect if this query contains a condition in specific columns. The second step is generating column statistics for each table as described in the previous section to collect metadata about each table. This metadata is collected from Java program take table data from text file as input then generate results in a text file as output. The third step is run JOMR by using a text file generated from the second step as input data to this algorithm after complete run algorithm generates new join order for this query. The last step is rewriting query by using new join order.

V. Benchmark (TPC-H)

To illustrate this approach, we used standard Benchmark [20, 21] so used the tables and join query of the TPC-H benchmark to test performance of the new join order and showed difference between original join order and new join order. The TPC-H benchmark content 6 tables joined together. We will test in three phases. The 1st phase with total number of rows around 8 million records distributed by The table lineitem contented 6,000,000 rows, Orders table contented 1,500,000 rows, PartSupp table contented 800,000 rows, Part table contented 200,000 rows Customer table contented 150,000 rows and Nation table contented 25 rows, 2nd phase with total number of rows around 85 million records distributed by The table lineitem contented 60,000,000 rows, Orders table contented 15,000,000 rows, PartSupp table contented 8,000,000 rows, Part table contented 2,000,000 rows Customer table contented 1,500,000 rows and Nation table contented 25 rows, and 3rd phase with total number of rows around 500 million records distributed by The table lineitem contented 300,000,000 rows, Orders table contented 75,000,000 rows, PartSupp table contented 40,000,000 rows, Part table contented 10,000,000 rows Customer table contented 75,000,000 rows and Nation table contented 25 rows.

VI. Experiment Result

We evaluate the effectiveness of JOMR on multi-join query, original Hadoop join order query.

A. Hardware

For our test environment we will evaluate JOMR in MapReduce framework by using three nodes. Each node with Intel core i5 2.4GHz processor, 4 GB of memory, 500 GB SATA disk and operating system windows 7 and each node generates three different data sets the first one with around 8 million records, the second one with around 85 million records, and last one with around 500 million records of TPC-H data.

B. TPC-H query and result

To evaluate JOMR algorithm we will test on TPC-H queries by using Q (3), Q (5) and Q (10). Now, we will describe a query structure. To generate column statistics we will use these queries.

Q (1): TPC-H Q(3)
Select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority
From customer c, orders o, lineitem l
Where c_mktsegment = 'BUILDING'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < '1995-03-15'
and l_shipdate > '1995-03-15';

Q (2): TPC-H Q(10)
Select c_custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment
From customer c, orders o, lineitem l, nation n
Where c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate >= '1993-10-01'
and o_orderdate < '1994-01-01'
and l_returnflag = 'R'
and c_nationkey = n_nationkey

Q (3): TPC-H Q(5) with some change
Select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue
From customer c, orders o, lineitem l, supplier s, nation n
Where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and o_orderdate >= '1994-01-01'
and o_orderdate < '1995-01-01';

In Fig. 4, shows the gap between cost of processing when use original query and JOMR algorithm in around 8 million records of TPC-H benchmark when using number of tables joined as factor to show the effects of our new algorithm with the increase of number of joined tables, we list the performance of query when use two, three, four, five and six tables to is showing results. Form reading the result found, the difference between original query in TPC-H and optimized query using JOMR when use two joined tables is typically cost time, when use three joined tables we found that JOMR enhanced the performance by around 20% but when we used six tables the performance by around 60% So our Experiments result showed that JOMR Improved the performance greater with increasing the number of joined Tables.

Fig 5: Query performance for 8 million records

In Fig. 5, compares the cost of processing by using two types of queries, the original query and optimized join order generated from JOMR in around 8 million records of TPC-H benchmark. From reading the result shows the optimized join order using JOMR has best performance.

Fig 6: Query performance for 85 million records

In Fig. 6, evaluates the join order generates from JOMR in around 85 million records of data. Still JOMR represents the
best performance of join plans using our query optimizer and shows that the performance increases when increasing the Data Size.

![Query number vs. Avg runtime (sec)](chart)

**Figure 7: Query performance for 500 million records**

In Fig. 7, evaluates the join order generated from JOMR in around 500 million records of data. Still JOMR represents the best performance of join plans.

### VII. CONCLUSION

In this paper, we have presented the new idea and implementation of our new algorithm, JOMR, for MapReduce. Given an SQL multi-join query, JOMR generates new join order of Multi-join query of Map-Reduce jobs, which minimizes the cost query processing. JOMR adopts how to generate minimum cost of multi-join query by using TSP, how to estimate cost between tables to evaluate and generate the best join order. We evaluated JOMR algorithm by using TPC-H tables and queries. Experiments results for JOMR algorithm proving the effectiveness of our query optimizer and this performance is improved more when increasing the number of joins and size of data.

### REFERENCES


