Cache Conscious Star-Join in MapReduce Environments

Guoliang Zhou
North China Electric Power University
zhouguoliang@ruc.edu.cn

Yongli Zhu
North China Electric Power University
yonglipw@heinfo.net

Guilan Wang
North China Electric Power University
wang.guilan@163.com

ABSTRACT

With the popularity of big data and cloud computing, data parallel framework MapReduce based data warehouse systems are used widely. Column store is a default data placement in these systems. Traditionally star join is a core operation in the data warehouse. However, little related work study star join in column store and MapReduce environments. This paper proposes two new cache conscious algorithms Multi-Fragment-Replication Join (MFRJ) and MapReduce-Invisible Join (MRIJ) in MapReduce environments. All these algorithms avoid fact table data movement and are cache conscious in each MapReduce node. In addition, fact table is partitioned into several column groups for cache optimization in MFRJ; One group contains all of foreign key columns and each measure column is a group. In MRIJ, each column is separately processed one by one which has higher cache utilization and avoids frequently cache miss from one column to the other column. MRIJ is composed of several map operation on dimension tables and one MapReduce job. We also apply MRIJ on RCFile in Hive. All operations are processed in mapping phase and avoid high cost of shuffle and reduce operation. If the dimension tables are big enough and cannot cache in local memory, MRIJ is divided into two phases, firstly each dimension table join with corresponding foreign key column in fact table as commonly map reduce join concurrently or serially; secondly all internal results joined for final results based on position index. This strategy also can be applied to other multi-table join. In order to reduce network I/O, dimension table and the fact table foreign key column are co-location storage. Our experimental results in cluster environments show that our algorithms outperform existing approaches in Hive system.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems

General Terms

ALGORITHMS, Performance

Keywords

MapReduce; Star join; Cache conscious; Column store

1. INTRODUCTION

With the rapid development of big data, conventional parallel data warehouse systems have been powerless because of high cost, poor scalability and unsatisfied fault-tolerant. So MapReduce-based [1, 2] data warehouse systems get more attention and applied extensively, such as Hive [3] in Facebook and Pig [4] in Yahoo. But those users often run into a performance problem. So, how to improve performance of the current MapReduce-based data warehouse systems is hot in both industrial and academic.

Data placement structure is one of the key factors for performance. In conventional data warehouse, column-store technologies [5–7] are used widely and high performance because they only read those attributes accessed by a query from disk and column-specific compression techniques. With the popularity of MapReduce framework, column store is introduced, such as RCFile [8] in Hive, COF [9] in IBM and CFile [10] in Llama. Based on these column store data structures, MapReduce algorithms performance is greatly improved.

In data warehouse, star schema is a most commonly used data model which is composed of a few big fact table and several small dimension tables. Star join is a core and important operation on star schema. But there is little related work attention to star join on similar column store in MapReduce based data warehouse system. Traditional join algorithms in MapReduce are proposed recently [10–12]. However, these algorithms do not consider the special situation of star join. In star schema, the fact table is very bigger than dimension tables and the fact table itself is also very large. When fact table stored in a distributed file system, critical to improve the performance of star join algorithms are to avoid fact table data movement or replication and processed locally. But conventional join algorithms cause fact table tuple movement and replication during shuffle and reduce phase significantly when applied to star join.

In addition, originally MapReduce is building on configuration low machine and the gap between cache and memory is smaller, so the cache feature is considered less. However, now the computer is highly configured in the cloud data center, and the gap between cache and memory is more and more. The cache characteristics will have a huge impact on performance [13]. Distributed caching technology has received much attention [14, 15]. To faster processing data in MapReduce environments, MapUpdate [17] and Main Memory Map Reduce(M3R) [18] are proposed which taking advantage of modern processor and memory technology. However, few MapReduce join algorithms consider cache and memory characteristics in each MapReduce node.

In this paper, we present two cache conscious star join algorithms full use of column store in MapReduce environments. One considers the column-group data placement, and the other each column separately storage.

The contributions of this paper are as follows:

1. We explore different fact table and dimension tables data placement in MapReduce environments for star join. With small dimension tables, dimension tables are replicated in distributed cache and the fact table is partitioned into foreign key column family and measure column families. For big dimension tables, dimension table and the corresponding fact table foreign key are co-location storage. With these data placement, disk and network I/O are effectively reduced.

2. We discuss several star join strategies on our data placement. MFRJ is a straight forward extension of Fragment-Replication Join for small dimension tables. MRIJ is divided into several map operation and one map-reduce join for small or big dimension tables. Each column is processed separately, reducing network traffic and I/O cost. This algorithm can also be applied to other multi-table join.

3. We do the experiments using SSB benchmark and compare the performance with the Hive. The results demonstrate the high performance of our star join algorithms.

The rest of the paper is organized as follows. We present detailed analysis of existing data placement and join algorithms on MapReduce framework in Section 2. We proposed MFRJ algorithm in Section 3 and MRIJ in Section 4. Performance evaluation is presented in Section 5. We conclude the paper in Section 6.

2. RELATED WORK

2.1 MapReduce and Data placement

MapReduce is a programming model for processing large data sets, and originally introduced by Google. The schema is composed of map and reduce two stages.

- Map: The master node divides the input into smaller sub-problems, and distributes them to worker nodes. The worker node processes the smaller problem, and passes the answer back to its master node.

- Reduce: The master node collects the results to all the sub-problems and sorted based on output key if needed. Then distributed the sorted data to worker node for computing the answer to the problem it was originally trying to solve.

MapReduce can utilize data locality processing data on or near the storage node to decrease data transmission. Hadoop [17] is an open-source implementation of MapReduce and widely used.

The difference of data organization format in MapReduce has a great influence on the amount of data transmission [10], and thus has a great impact on performance. So MapReduce based data placement is concerned. A few new data placement structures are proposed such as RCFile, Cfile, etc. In conventional data warehouse system, three data placement structures are proposed, which are row-store, column-store and PAX [7] storage. Each structure has its own advantages. In short, row-store is write-optimized, column-store read-optimized and PAX compromising read write operation. With the popularity of MapReduce, similar data structures are proposed. Corresponding relations are shown below in Table 1.

Table 1: Data storage format in MapReduce.

Data Placement	MapReduce	System
Column-store	Bigtable[2]	Google
	COF[9]	HBase
	CFile[10]	Llama
PAX[7]	RCFile[8]	Hive, Pig

Among them, column-store is preferred to in MapReduce as in conventional data warehouse. Commonly all columns are partitioned into several column families for spatial locality and reducing cost of tuple construction. In some other cases, each column is separately storage. Now RCFile (Record Columnar File) is a default option in Hive and Pig systems. The developers can use *STORED AS RCFile* clause creating RCFile format file. RCFile partitions the table first horizontally into some group or block, then blocks are vertically split into several column families.

2.2 Join algorithms in MapReduce

Initially MapReduce is used to process one data set, so join operation is not necessary. With the expansion of application scope, MapReduce need analysis multi data sets, and join operations are introduced. Join is a common and important operation in relational database and well studied. In MapReduce environment, several join strategies are supported [10-12, 18].

- Map Reduce Join

  This is the most common approach to do join operation. Two tables are partitioned based on join key in map phase and sorted based on the key in shuffle phase. Each partition is joined at the corresponding node in reduce phase. This approach is like hash-join in a traditional database.

- Map Side Join

  If one of the two tables is small enough, the join can be completed in map phase. The query does not need a reducer. Small table is distributed in local memory for each mapper node. Then a fragment of big table joins with small table in each mapper. This approach is referred to Fragment-Replication Join [10].

- Bucketed Map Join

  If tables being joined are bucketized, and the buckets are a multiple of each other, the buckets can be joined with each other. Join can be done on the mapper only.

This approach is similar to the data preprocessing [18] and sort-merge join.

2.3 Star-join in MapReduce Environments

Star-join is a very common operation in data warehouse. The star-join consists of one fact table F referencing several dimension tables $D_1, D_2, ..., D_n$. Throughout this article, we use the following definitions:

- D_i has the primary key PK_i that is associated with the foreign key FK_i of F where i is the dimension identification number of D_i
- Fact table has format: $F(fk_1, fk_2, ..., fk_n, m_1, m_2, ..., m_k)$ where fk_i is the value of the foreign key FK_i and m_i is a measure values.
- The star-join query might have restrictions CD_i on D_i and CF_i on F.

Generally, star-join has the following form:
SELECT D1.val, D2.val, F.val
FROM D1 JOIN F ON (D1.pk1 = F.fk1)
JOIN D2 ON (D2.pk2 = F.fk2)
WHERE CD1 AND CF1
In order to more effectively deal with star join, concurrent join [10] and Scatter-Gather-Merge [12] algorithms were proposed. In concurrent join, a query is split into sub-queries over multiple datasets. These sub-queries are executed concurrently by distinct MapReduce phases. This strategy can reduce network overhead and I/O costs. When applied to star-join, for example star join $F \bowtie D_1 \bowtie D_2$ is divided into $F \bowtie D_1$ and $F \bowtie D_2$. In reduce phase, the two temporary results are joined. Scatter-Gather-Merge join has the similar idea of concurrent join. Fact table is partitioned according to the dimension table and joined for each partition, and then the intermediate results are joined.

However, all these algorithms will cause a big intermediate result and high I/O cost. Intermediate result need distributed again and high data replication cost.

In addition, now star join is supported in Hive. For the above star join, two map/reduce jobs involved in computing the join as the format $((F \bowtie D_1) \bowtie D_2)$. The first of these joins D_1 with F and buffers the values of D_1 while streaming the values of F in the reducers. The second of these jobs buffers the results of the first join while streaming the values of D_2 through the reducers. We think this is a very inefficient way for high I/O and data replication between two MapReduce jobs.

3. MULTI-FRAGMENT-REPLICATION JOIN

3.1 Multi-Fragment-Replication Join and Data placement

In a star schema model, dimension tables are smaller and the fact table is bigger commonly. The core idea of improving the performance is reducing the cost of data replication and movement for the fact table. Fact table is localization processing. Dimension tables can be replicated or moved. So we can complete star join like Fragment-Replication Join. Firstly, we cache all dimension tables in local memory for each mapper. Then streaming the value of fact table joins with all dimension tables. That is called our Multi-Fragment-Replication Join (MFRJ).

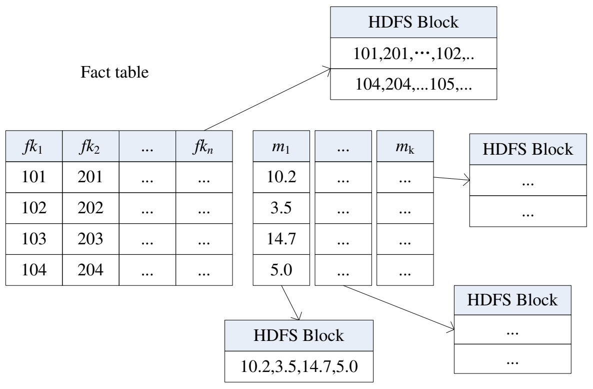


Figure 1: The fact table data storage format

But fact table different data organization format has a great influence on the performance. If the table is row store, many measure columns are read but not used, so cache utilization is low. If the table is full column store that is each column separately storage, frequently reading data from one column to the other column will occur serious cache and TLB(Translation Lookaside Buffer) miss.

In star schema, the number of measure columns is more than foreign key(*fk*) columns, and *fk* columns have higher operating frequency. Moreover, star join requires access to majority of *fk* columns in most case. Utilizing column family data placement, we put all foreign key columns in one group and each measure column in each other one group. Fact table storage structure is shown as in Fig 1.

In this data placement, cache utilization can be represented by the following formula.

$$C_u = \sum_{i=1}^{s} size(fk_i) / \sum_{i=1}^{n} size(fk_i) \approx 1 \quad (1)$$

Where s is the number of dimension tables associated with star join.

MFRJ algorithm is completed through the following stages.

1. We distribute and cache all dimension tables in each mapper node. For decreasing local memory needed, based on column-store we can only distribute related columns of dimension table in star join.
2. Map operation

   For each row in fact table joined with all dimension tables between primary key and foreign key. The output key-value like $\{position, < v_1, v_2, .., v_n >\}$, where $position$ is like $rowid$ for identify one row, v_i is the value draw from dimension table for $SELECT$ clause needed columns. For each measure column, we apply query condition and output key-value like $\{position, < m_i >\}$. If the query dose not related to measure column, then the reduce phase is not needed.
3. Reduce operation

   All input are sorted based on position key. If the count of element in each key is equal to the sum of 1 and number of measure columns, then all values of this key compose of output like $< v_1, v_2, .., v_n, m_i >$. That is the final results of star join.

3.2 Optimization and evaluation

In some case, dimension tables are big enough and the cost of caching and replication is heavily. We choose those smaller dimension tables replicated and join the fact table firstly, then the result joins big dimension tables like conventional map reduce join algorithms.

In MFRJ algorithm, all foreign keys are processed simultaneously. For each foreign key column, we need to read a dimension table join with. If the size of all dimension tables is bigger than local memory, dimension tables are frequently swapped in out and result in cache-thrashing. This will seriously affect the efficiency of our algorithm. So, local memory of each mapper node has the following requirements.

$$M >= \sum_{i=1}^{n}(size(row_{D_i}) \times |D_i|) \tag{2}$$

Where M is the size of local memory in mapper node.

Usually, performance of join algorithms in a computer cluster is determined by network and disk I/O. The amount of data movement in MFRJ is mainly dimension table distributed in map phase and filtered key values in shuffle phase.

$$\begin{aligned} M_T \approx m \times \sum_{i=1}^{n}(size(row_{D_i}) * |D_i|)+ \\ \prod_{i=1}^{n} f_i \times |F| \times size(row_{f\_group}) + |F| \times size(m_i) \end{aligned} \tag{3}$$

Where m is number of node in MapReduce, n is the numbers of star join related dimension tables, $f_0, ..., f_i$ donte the selectivity of filter for each dimension table.

In addition, disk I/O of MFRJ is mainly caused by accessing fact table.

$$M_D \approx |F| \times size(row_{f\_group}) \tag{4}$$

However, in some column store data warehouse each column is storage respectively [9]. MFRJ algorithm is not inappropriate for frequently cache miss from one to the other column. In addition, MFRJ algorithm has critical memory requirement and some node cannot meet this. So we proposed MapReduce-Invisible Join on separately storage structure and small memory node.

4. MAPREDUCE-INVISIBLE JOIN

4.1 Procedure of MapReduce-Invisible Join

In some column store data warehouse, each column is respectively storage in different data node. So we process each column one by one for data locality. This way is like invisible join [6]. So we called the algorithm as MapReduce-Invisible Join. In this algorithm, star join can be completed by several small map operation and one map reduce join. The procedure is divided into the following steps:

1. Filter dimension tables

   Choose dimension table $D_1, D_2, ..., D_n$ in query and apply the query condition. Then key-value $dkv_1, dkv_2, ..., dkv_n$ are generated, in each key-value, the key is primary key value of this dimension table, the value is the other needed column value of this dimension table. The $dkv_1, dkv_2, ..., dkv_n$ are replicated to each map node.

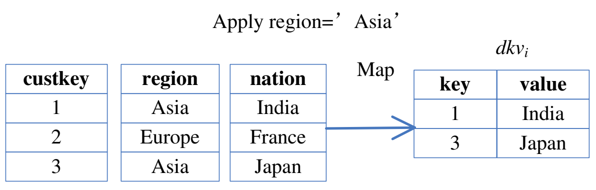


Figure 2: Filter dimension table

2. Map operation

   Each column fk_i in fact table joins with corresponding key-value dkv_i like Fragment-Replication Join. The result is another key-value fkv_i, where key is position of fact table and the value is from dkv_i.

   The measure columns of fact table are also processed like above, and generate key-value like fkv_i.

3. Reduce operation

   All input key-values fkv_i are shuffled and sorted based on the *position* key. For each key if the count is equal to $n + m$, then output and gets final result, where m is number of measure columns related to query.

We use an example Query 3.1 in SSB (Star Schema Benchmark) [16] to illustrate this process of MRIJ. This query is composed of one fact table and three dimension tables. Each dimension table has one filter condition. The result includes four columns which three from three dimension tables and one measure from the fact table. In order to better illustrate the procedure, we add a query condition on measure column. The star join query is below.

```
SELECT c_nation, s_nation, d_year,lo_revenue
FROM customer
JOIN lineorder ON lo_custkey = c_custkey
JOIN supplier ON lo_suppkey = s_suppkey
JOIN ddate ON lo_orderdate = d_datekey
WHERE c_region = 'ASIA'
AND s_region = 'ASIA'
AND d_year >= 1992 and d_year <= 1997
AND lo_revenue > 40000;
```

- Processing dimension tables query conditions

  Each query condition is applied to the corresponding dimension table and output a hash table that satisfies the predicate. The key is dimension table primary key and the value is using column in query. This operation is completed by a Map operation. All involved dimension tables are processed parallel. We save the result in each local node cache (for example Hadoop Distributed Cache) for reuse in the next phase. As in Fig 2, we read three columns from the customer table and apply query condition *region='Asia'* on region column. If the condition is satisfied , then this record is put into hash table. Same operations are applied on *supplier* and *ddate* dimension tables. At the same time, all three jobs can be executed concurrently and three dkv_i are generated.

- MapReduce procedure in MRIJ

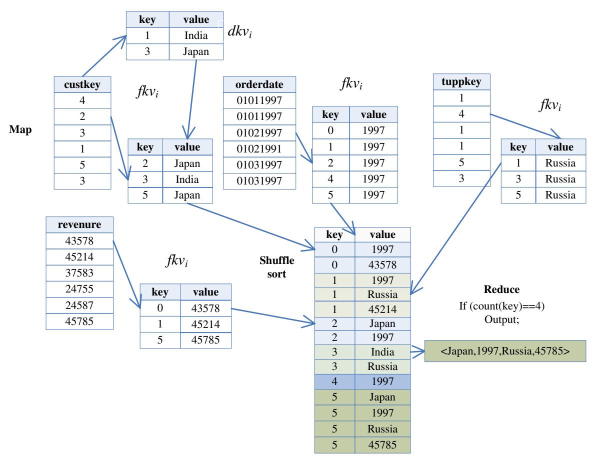


Figure 3: Process of MR-Invisible-Join

Each dkv_i is separately joined with corresponding fk column and generate fkv_i. Measure columns are processed similar. The entire process is shown in Fig 3.

In MRIJ, fact table is processed locally and not data replication and movement. Each column is separately read and joined, cache utilization is high. The size of data transfer is about:

$$M_T = m \times (\sum_{i=1}^{n} size(dkvi) + \sum_{i=1}^{n} size(fkv_i)) \qquad (5)$$

Where m is number of nodes in MapReduce, n is the numbers of star join related dimension tables.

Local memory of each mapper node has the following requirements.

$$M = max(size(dkv_i)) \qquad (6)$$

4.2 MapReduce-Invisible Join on RCFile

RCFile is a default data storage format in Hive. A table stored in RCFile is first horizontally partitioned into multiple row groups. Then, each row group is vertically partitioned so that each column is stored independently. MRIJ algorithm is composed by several phases on RCFile format.

1. Same as MRIJ algorithm.
2. For the first foreign key column of fact table, each value joined with dkv_1, the result is key value fkv_1 has the format $\{position, < v_1 >\}$, where $position$ is position information and v_1 from dkv_1.
3. Choose next dkv_i and join with corresponding column fk_i which in fkv_{i-1} position list, and generate fkv_i like step 2.
4. Repeat step 2 and 3 until all dkv_i are processed.
5. Based fkv_n position list fetch measure column in join query.

In RCFile block, each hash table is separately used to get the position of fact table that satisfy the query condition one by one. For each foreign key column in the fact table, we get a key value pair where key is position and value is column value of the dimension table. An example is showed below in Fig 4.

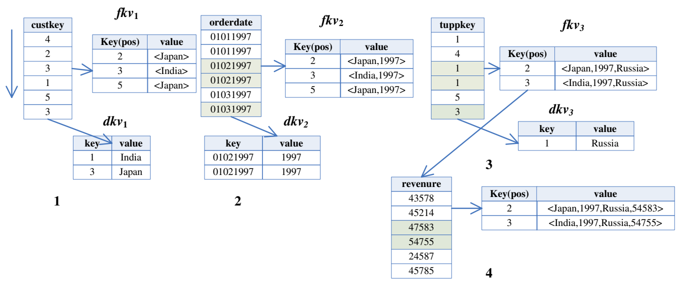


Figure 4: Process of MP-Invisible-Join on RCFile block

The algorithm's main purpose is to take full advantage of the locality of the fact table data to avoid network traffic during join processing and enhance the star-join query processing efficiency. All phases are completed in mapper and avoid high cost of shuffle operation. Data transmission of MP-Invisible-Join algorithm on RCFile is about:

$$M_T = n \times (\sum_{i=1}^{m} dkv_i) \qquad (7)$$

Where n is number of node in Cloud, and dkv_i is hash table constructed on dimension table D_i.

4.3 MapReduce-Invisible Join with big dimension tables

In some case, dimension tables are large enough and cannot be storage in the local memory of each map node. Therefore, we deal with star join through two MapReduce jobs:

- First, each dimension table separately joins with corresponding fact table foreign key column. The result is key-value $\{pk_i, < values, position >\}$, where $values$ are columns value involved in the query of dimension table and $position$ is position in fact table.
- Second, in map step, each $\{pk_i, < values, position >\}$ is transformed to $\{position, < values >\}$, at the same time the related measure columns are also input as $\{position, < measure >\}$ format. In shuffle phase, all intermediate results sort based on $position$ key. In reduce step, if each position counter is equal to the sum number of dimension tables involved in the join and measure columns, then selecting needed column output, to get the final result.

The process can be represented by the following formula and execution plan as Fig 5.

$$\begin{aligned} R = (pk_1 \bowtie D_1) \bowtie (pk_1 \bowtie D_1) \bowtie ... \bowtie (pk_1 \bowtie D_1) \\ \bowtie (m_1, ..., m_i) \end{aligned} \qquad (8)$$

In this algorithm, as we can see, star join is partitioned into several small joins. Performance of the algorithm is proportional to the number of dimension tables.

Similarly, in this algorithm the first job is costly. In this phase, fact table foreign key column and corresponding dimension table are not in the same MapReduce node and

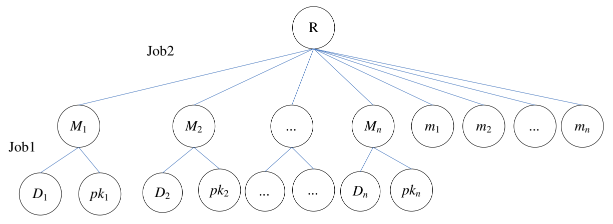


Figure 5: Execution plan of MRIJ with big dimension tables

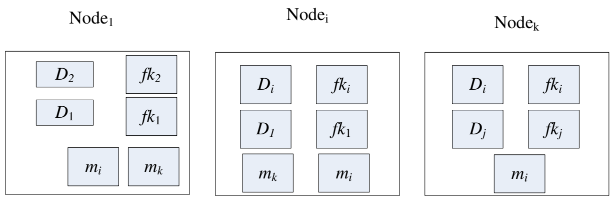


Figure 6: Dimension tables and fact table storage with co-location

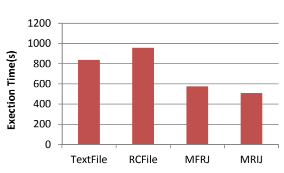
occur high network I/O. We solved the problem by implementing a new HDFS block placement policy. Dimension table and related foreign key column are co-location storage as Fig 6. So, fact table is processed locally.

5. PERFORMANCE EVALUATION

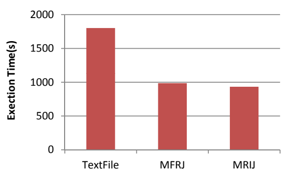
In this paper, we use the SSB benchmark comparing the performance of our algorithms and Hive. The SSB benchmark schema is composed of one fact table (LINEORDER) and four dimension tables (CUSTOMER, SUPPLIER, PART and DATE). SSB consists of thirteen queries which divided into four categories from simple to complex. With the scale factor *SF* changed, size of LINEORDER table is *SF**6000000. We set *SF*=10 for adapting to the experimental environment and fact table is about 6GB in that case. We adopt TextFile and RCFile data storage in Hive. In our test, all dimension tables and the fact table are storage in HDFS.

We use Hadoop version 0.20.2 and configured it to run up to three map and reduce tasks per node. We use Hive version 0.8.1. For the MapReduce environment, we used 6 nodes, and the name node is DELL PowerEdge T310 with Intel(R) Xeon(R) CPU X3430@ 2.40GHz and 2GB RAM running Linux Ubantu version. Five data nodes are Lenovo computer with Intel Pentium D processor and 2GB RAM. All nodes are connected through a switched 1 Gbps Ethernet LAN.We choose SSB query 3.1 and 4.1 as our tasks. We only compute join operation and remove the group by clause. We study the performance and scalability of our algorithms with respect to different query format. Query 4.1 is composed of one fact table and four dimension tables, and the query is changed as below for our test.

SELECT c_nation, p_name, s_nation, d_year, lo_revenue, lo_supplycost
FROM lineorder
JOIN customer ON lo_custkey = c_custkey
JOIN supplier ON lo_suppkey = s_suppkey
JOIN part ON lo_partkey = p_partkey
JOIN dwdate ON lo_orderdate = d_datekey

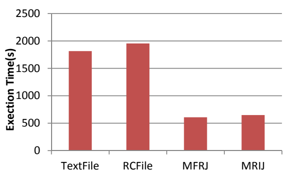


(a) Query 3.1

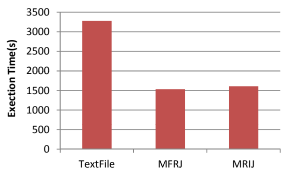


(b) Query 4.1

Figure 7: Original Query



(a) Query 3.1



(b) Query 4.1

Figure 8: No where conditions

WHERE c_region = 'ASIA'
AND s_region = 'ASIA'
AND d_year >= 1992 and d_year <= 1997
AND (p_mfgr = 'MFGR#1' or p_mfgr = 'MFGR#2');

In some case, MapReduce jobs of Hive translating from HQL are inefficient, so SQL-to-MapReduce translator is proposed such as YSmart [20, 21]. However, we do not compare our join Map/Reduce program with YSmart generated code because YSmart dose not consider data placement.

5.1 SSB query 3.1 and 4.1

We first compare original SSB Q3.1 and Q4.1 with different join algorithms. The results are shown in Fig 7. As we have seen, Hive join algorithms have similar performance in TextFile and RCFile in Fig 7(a). This is mainly because although RCFile is column storage, but RCFileInputFotmat is row-oriented. Thus, advantages of column store do not play a role. Our experimental results also show that Hive join on TextFile and RCFile has similar I/O and performance. At the same time, our algorithms obtained approximately twice the acceleration because of less disk I/O and cache feature optimization. In star join, the primary cost is disk I/O of the fact table. Our algorithm needs to read fact table data are approximately half of the HIVE algorithm.

5.2 no where condition

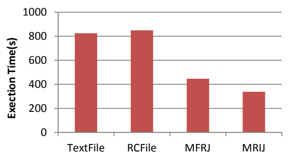
In some case, the star join only include join condition no where condition, we test Q3.1 and Q4.1 by removing where condition on dimension tables. The result is shown in Fig 8. In this case, our algorithm gets twice speedup. The experimental results coincide with the original query.

5.3 not include measure columns

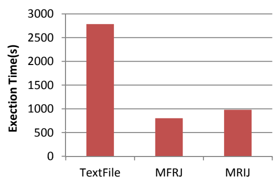
In our algorithms, measure columns are separately processed. If the star join does not include measure columns, our algorithms can acquire higher speedup. MFRJ can be completed in map phase and MRIJ does not require access measure columns. The result is shown in Fig 9.

5.4 MRIJ with big dimension tables

If the dimension tables are big enough, distributed di-



(a) Query 3.1



(b) Query 4.1

Figure 9: No measure columns

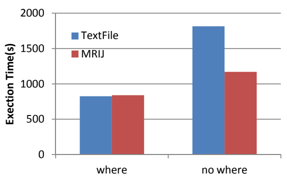


Figure 10: MRIJ with big dimension tables

mension tables cost is heavy. So our MRIJ algorithms are partitioned into two phases. As we can see, MRIJ performance is similar or better than Hive join in Fig 10 of query 3.1. In addition, MRIJ is proportional with the number of dimensions. So, In the higher dimensional case, the MRIJ algorithm has higher performance.

6. CONCLUSIONS

In this paper, we present two star join algorithms in MapReduce framework. All these algorithms apply column-wise partitioning scheme to improve cache characteristics. We study the performance problem on different data placement. We evaluate algorithm's performance by comparing it against Hive on SSB datasets and provide the speedup of two times compared to Hive.

In next step, we will apply data compression in column storage for less I/O and high performance. We will also study cache and network conscious algorithms in MapReduce environments.

Acknowledgments

Research was sponsored by The Higher Education Scientific Research Project of Hebei Province under Grant No. Z2011306.

7. REFERENCES

[1] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, in OSDI, 2004, pp. 137-150.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, etc. Bigtable: A Distributed Storage System for Structured Data. OSDI '06 pp. 205-218

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu,P. Wyckoff, and R. Murthy, Hive - a warehousing solution over a Map-Reduce framework, PVLDB, vol. 2, no. 2, pp. 1626-1629, 2009.

[4] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston,B. Reed, S. Srinivasan, and U. Srivastava, Building a high level dataflow system on top of MapReduce: The Pig experience, PVLDB, vol. 2, no. 2,pp. 1414-1425, 2009.

[5] M. Stonebraker, D. J. Abadi, A. Batkin, etc., C-store: A column-oriented DBMS, VLDB 2005, pp. 553-564.

[6] D. J. Abadi, S. Madden, and N. Hachem, Column-stores vs. row-stores:how different are they really? SIGMOD 2008, pp. 967-980.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, Weaving relations for cache performance, VLDB 2001, pp. 169-180.

[8] Rubao Lee, Yin Huai, Zheng Shao, etc. RCFile: A fast and space-efficient data place-ment structure in MapReduce-based warehouse systems. ICDE 2011, pp. 1199 - 1208

[9] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 2011. Column-oriented storage techniques for MapReduce. Proc. VLDB Endow. 4, 7 (April 2011), 419-429.

[10] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, Sai Wu: Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. SIGMOD 2011, pp. 961-972

[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita,and Y. Tian. A comparison of join algorithms for log processing in mapreduce. SIGMOD 2010, pp. 975-986

[12] Hyuck Han, Hyungsoo Jung, Hyeonsang Eom, Heon Young Yeom: Scatter-Gather-Merge: An efficient star-join query processing algorithm for data-parallel frameworks. Cluster Computing 14(2): 183-197 (2011)

[13] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. VLDB 1999, pp. 78-89.

[14] Brewer EA. Towards robust distributed systems (abstract). PODC 2000, pp. 7.

[15] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng, Accelerating MapReduce with Distributed Memory Cache, ICPADS 2009, December 8-11, 2009, Shenzhen, China

[16] P. E. O'Neil, E. J. O'Neil, and X. Chen. The Star Schema Benchmark (SSB). http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[17] Apache Hadoop. http://hadoop.apache.org/, 2012

[18] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. 2012. Muppet: MapReduce-style processing of fast data. Proc. VLDB Endow. 5, 12 (August 2012), 1814-1825.

[19] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. 2012. M3R: increased performance for in-memory Hadoop jobs. Proc. VLDB Endow. 5, 12 (August 2012), 1736-1747.

[20] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. YSmart: Yet another SQL-to-MapReduce Translator. ICDCS 2011, Minneapolis, Minnesota, June 20-24, 2011.

[21] Yin Huai, Rubao Lee, Simon Zhang, Cathy H. Xia, and Xiaodong Zhang. DOT: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. SOCC 2011, Cascais, Portugal, October 27-28, 2011.