An Efficient Theta-Join Query Processing Algorithm on MapReduce Framework

Shih-Ying Chen
Department of Computer Science and Information Engineering
National Taichung University of Science and Technology
sychen@nutc.edu.tw

Tsui-Ping Chang
Department of Information Technology
Ling Tung University
apple@ltu.edu.tw

Zhi-Hong Chang
Department of Computer Science and Information Engineering
National Taichung University of Science and Technology
s18993111@nutc.edu.tw

Abstract**—**As the rapid development of hardware and network technology, cloud computing has become an important research topic. For applications of large-scale data processing, such as data warehouse, MapReduce is the most famous platform for parallel data processing in cloud computing. To support the star-join queries in data warehouse, Scatter-Gather-Merge (SGM) proposes an efficient algorithm on the MapReduce framework. However, SGM supports only the equi-join queries. Nonequi-join queries may cause SGM to fail. In this paper, we propose a method to cope with theta-join queries, i.e., both equi-join and nonequi-join queries. Our proposed method uses a novel manipulation of keys for partitioning data. The key manipulation matches up the MapReduce paradigm, and makes theta-join queries workable on the MapReduce platform. Our experimental results show that the proposed method achieves similar performance to SGM, but our method supports more join-query types. Our method performs even better than SGM in some query types of high data selectivity.

Keywords: theta-join queries, query processing large-scale data, MapReduce platform

I. Introduction

As the success of the Google File System (GFS) [5] and the Hadoop Distributed File System (HDFS) [9], cloud computing has become an important research topic in recent years. The Hadoop Distributed File System [9], which is inspired by the Google File System [5], contains one NameNode and a number of DataNodes. When storing a file, the HDFS splits the file into several blocks of the same size and stores them to the DataNodes.

MapReduce [4] is a programming model, which is useful for processing large-scale data in parallel. A MapReduce program consists of a Map function and a Reduce function. One master node invokes a number of workers to complete a MapReduce program. Suppose that a user stores input data in the HDFS file system and runs a MapReduce program. The input data is represented as a set of <*key*, *value*> pairs. The master invokes M Mappers (i.e., the nodes executing a Map function) and sends parts of the input data to them. Later on, R Reducers (i.e., the nodes executing a Reduce function) will be invoked to complete the program. By using the *key* values, each Mapper partitions its given data into R region files and stores them on the Mapper itself locally. Once all the Mappers finish their works and notify the master, the master will invoke R Reducers. The R Reducers read their corresponding region files stored on the Mappers, and then sort the <*key*, *value*> pairs by the *key* value to generate another set of <*key*, *list*(*value*)> pairs by the program's logic. Finally, the collection of the set of *list*(*value*)s on every Reducers is the result of the MapReduce program.

MapReduce provides a new platform to process data in parallel, especially for large scale data. Many MapReduce programs [1, 2, 3, 8] are developed to deal with the star join queries for large scale data in data warehouse. A star join query consists of some dimension tables joining with fact tables (usually one fact table). The large scale of data makes star join queries more time-consuming. Existing MapReduce star-join algorithms [2, 3] only consider equi-join queries. None of them can support theta-join (i.e., both equi-join and nonequi-join) queries.

In this paper, we propose a MapReduce method, which supports star join queries with theta-join operations. The contributions of this study are as follows. A method on the MapReduce framework is proposed and implemented for star joins with theta-join operations. The method also offers a novel manipulation of keys for partitioning data. The key manipulation matches up the MapReduce paradigm, and makes theta joins workable on the MapReduce platform. Experiments are also conducted for evaluating the performance of the proposed method.

The paper is organized as follows. Section II presents the proposed method. Section III illustrates the experimental results. Section IV is our conclusions and future work.

II. Proposed Method

Since the MapReduce methodology uses <*key, value*> pairs to represent data, the fact table and the dimension tables for a star join query are represented as follows. For the dimension table D_i, its schema is represented by $D_i(PK_{Di}, V_{Di})$, where PK_{Di} denotes the primary key of D_i and V_{Di} denotes the other columns that excludes PK_{Di}. Therefore, each tuple of D_i is represented by $<pk_i, v_{Di}>$ pairs, where pk_i is the value of the primary key PK_{Di}, and v_{Di} denotes the collection of the column values of V_{Di}. Similarly, the schema of the fact table F is represented by $F(FK_1, FK_2, ..., FK_n, V_F)$, where FK_i is the foreign key referring to D_i, and V_F denotes the rest columns of F. Each tuple of the fact table F is represented by $<(fk_1, fk_2, ..., fk_n), v_F>$ pairs, where fk_i is the value of FK_i and v_F denotes the collection of column values of V_F. On the other hand, for a join query, R_{Di} and R_F denote the non-primary-key columns of table D_i and table F specified in a SELECT clause, respectively. And, r_{Di} and r_F denote their values. The join operator between the fact table F and the dimension table D_i is denoted by C_i, which can be one of the operators =, <, >, ≥, ≤ and ≠.

MapReduce provides a default partition function, however, our proposed method designs a key partition function for the purpose of supporting theta-join operation and enhancing join performance. The key partition function makes the fact table joining with only one dimension table at each worker. As a result, the join performance can be improved. To achieve this, a novel manipulation on keys to generate MapReduce's <*key, value*> pairs is offered in our method.

Our proposed method, namely Theta-Join Scatter-Gather Merge (TJSGM), consists of three phases with Map and Reduce functions in each phase. They are the filter phase, scatter-and-gather phase, and the merge phase.

The filter phase is used for reducing the data sizes for processing, as shown in Algorithm 1. Tuples of dimension tables are filtered according to the filter conditions of queries. The input of Algorithm 1 is the tuples of dimension tables that is going to be filtered, and the output is the filtered result. The Map function in the phase reads the tuples $<pk_{Di}, v_{Di}>$ of dimension tables predicated with filter conditions, and generates $<(pk_{Di}, i, C_i, “D”), r_{Di}>$ pairs in lines M1 and M2. Note that the keys of the generated pairs are reformatted to $(pk_{Di}, i, C_i, “D”)$ rather than merely pk_{Di}. The flag "D" indicates the pairs come from dimension tables. The Reduce function then reads the pairs to generate a bloom filter file for each dimension table predicated with filter conditions (lines R1and R2). The bloom filter files consist of the primary keys pk_{Di} only.

```
Algorithm 1:  Filter Phase
Map() {
    Input: $<k, v> = <pk_{Di}, v_{Di}>$, where dimension table $D_i$ is
    going to be filtered
    Output: $<(pk_{Di}, i, C_i, “D”), r_{Di}>$ pairs
    M1:   if ($v_{Di}$ satisfies the filter condition of $D_i$)
    M2:       Emit $<(pk_{Di}, i, C_i, “D”), r_{Di}>$
}
Reduce() {
    Input: $<(pk_{Di}, i, C_i, “D”), r_{Di}>$ pairs
    Output: $<(pk_{Di}, i, C_i, “D”), r_{Di}>$ pairs and a bloom filter
    $BF_i$ of $D_i$.
    R1:  Emit $<(pk_{Di}, i, C_i, “D”), r_{Di}>$
    R2:  Add $pk_{Di}$ to $BF_i$
}
```

In the Scatter-and-Gather phase, joins on the fact table and the dimension tables are actually performed, as shown in Algorithm 2.

There are three cases that the Map function emits data to the Reduce function. In the first case, each key values of pairs of the fact table F contained in the corresponding bloom filter are emitted, as shown in lines M1 to M10. Please note that, in line M9, the emitted key of a pair is $(fk_i, i, C_i, “F”)$, and its value is $(fk_1, fk_2, …, fk_n, r_F)$. The flag "F" indicates these pairs come from the fact table. In the second case, tuples of dimension tables not being filtered in the filter phase are emitted, as shown in lines M12 and M15. Please note that, the emitted key of a pair in this case is $(pk_{Di}, i, C_i, “D”)$, and its value is r_{Di}. The flag "D" indicates these pairs come from the dimension table. In the third case, the output pairs from the filter phase in Algorithm 1 are emitted directly, as shown in lines M16 to M18. Note that in the three cases the Map function manipulates the keys on purpose for accomplishing theta-join operation in the next Reduce function.

The Reduce function in Algorithm 2 is used for joining purpose. Once a Reducer reads the emitted data pairs from the Mappers, it deals with two cases: equi-join and non-equi-join operators. In the first case of equi-join, the Reduce function merges the pairs of dimension tables and fact table, and emits them, as shown in lines R1 to R4. In the second case, the Reduce function stores all the required data from the dimension tables and the fact table, as shown in lines R5 to R 15. In the function, DTK, DTV, FTK, and FTV are lists for storing dimension tables' keys, dimension tables' values, the fact table's keys, and the fact table's values, respectively. After obtaining all the required data, the result which satisfies the comparator C_i are sent to next phase, as shown in lines R16-R22.

```
Algorithm 2: Scatter-and-Gather Phase
Map() {
    Input: $<k, v>$ and bloom filter files $BF_i$, where $<k, v> =$
        $<(fk_1, fk_2, ..., fk_n), v_F>$ of table $F \cup <pk_{Di}, v_{Di}>$ of table
        $D_i$ not being filered $\cup <(pk_{Di}, i, C_i, “D”), r_{Di}>$ from
        Algorithm 1
    Output:  a set of pairs, $T$
  M1:   T=null
  M2:   case  1:$v$ is a record of the fact table F {
  M3:     for each $fk_i$ in $v${
  M4:         if ($fk_i \notin BF_i$) {
  M5:             Exit map()
  M6:         }
  M7:     }
  M8:     for each $fk_i$ in $v$ {
  M9:         T=T $\cup <(fk_i, i, C_i, “F”), (fk_1, fk_2, …, fk_n, r_F)>$
  M10:  }
  M11: }
  M12: case  2:$v$ is a record of the dimension table {
  M13:     Reformate $<pk_{Di}, r_{Di}>$ into $<(pk_{Di}, i, C_i, “D”),$
           $r_{Di}>$
  M14:     T=T$\cup<(pk_{Di}, i, C_i, “D”), r_{Di}>$
  M15: }
  M16: case  3:$v$ is a record from Algorithm 1 {
  M17:     T=T$\cup <(pk_{Di}, i, C_i, “D”), r_{Di}>$
  M18: }
  M19: Emit T
}
Reduce() {
  Input: <k, v>, where k is a key consisting of ($pk_{Di}$, i, $C_i$, flag), and v is a set of records that have same join key.
  Output: <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)> pairs
  R1:   if ($C_i$ is "=") {
  R2:         Make an output <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
  R3:         Emit <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
  R4:   }
  R5:   if ($C_i$ is not "=") {
  R6:         for each v {
  R7:           if($v \in F$) {
  R8:             Add k to FTK and Add v to FTV
  R9:           }
  R10:          if($v \in D_i$ for some table $D_i$) {
  R11:            Add k to DTK and Add v to DTV
  R12:          }
  R13:        }
  R14:   }
  R15: }
Cleanup() { //do after Reduce() complete
  R16:    for each FTK {
  R17:      for each DTK {
  R18:        if (FTK and DTK satisfies $C_i$) {
  R19:          Emit <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
  R20:        }
  R21:      }
  R22:    }
}
```

In the Merge phase, the result of joining the fact table and the dimension tables are merged, as shown in Algorithm 3. The input is the result of the Scatter-and-Gather phase (i.e. the result of joining the dimension tables and fact table). And the output is the result of data after merging.

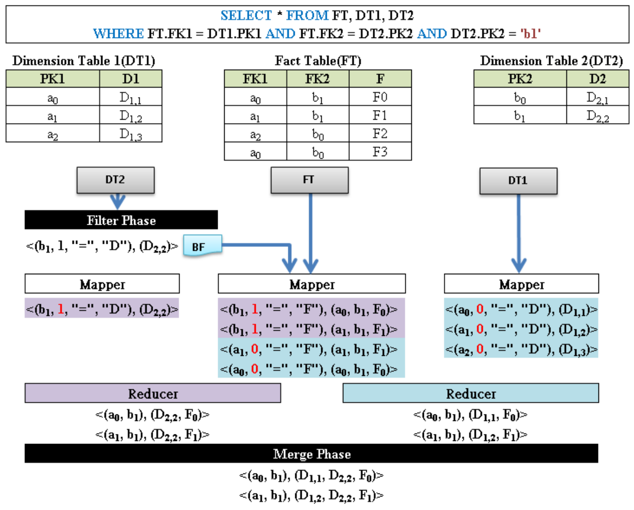
The Map function emits the key-value pairs of the previous phase to the Reduce function directly in line M1. The Reduce function then merges the result in lines R1 and R2.

```
Algorithm 3: Merge Phase
Map() {
  Input: <k, v>, where k is ($fk_1$, $fk_2$, …, $fk_n$), and v is ($r_{Di}$, $r_F$).
  Output: <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
  M1:  Emit <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
}
Reduce() {
  Input:<k, v>= <($fk_1$, $fk_2$, …, $fk_n$), ($r_{Di}$, $r_F$)>
  Output: <($r_{D1}$, $r_{D2}$, …$r_{Dn}$), $r_F$)>
  R1:  Merge v to collect all fields of factor table and
       dimension tables
  R2:  Reformat and Emit <($r_{D1}$, $r_{D2}$, …$r_{Dn}$), $r_F$)>
}
```

Figure 2 illustrates an example of TJSGM. There are three tables, namely *DT1*, *DT2*, and *FT*. Table *DT1* has primary key *PK1* and the other set of columns, which is represented by *D1*for simplicity. Table *DT2* uses similar denotations to *DT1*. Table *FT* has foreign keys *FK1* and *FK2* referring to *PK1* and *PK2*, respectively. Data tuples of the tables and the SELECT clause for our example of join query are shown in Figure 2.

During the Filter phase, *DT2* is filtered by Algorithm 1, since there is a filter condition on *DT2* in the SELECT clause. The generated set of key-value pair is <(b_1, 1, "=", "D"), $D_{2,2}$>, where(b_1, 1, "=", "D") is the key value. In addition, bloom filter files are generated for later use.

During the Scatter-and-Gather phase, the map function emits key-value pairs from *DT1*, *FT*, as well as the pair <(b_1, 1, "=", "D"), $D_{2,2}$> from the filter phase. Please note that the values of keys are generated purposely for joining in this phase 2. Our key partition function is applied in the phase. The Reduce function in the phase to partially complete the join operation. After the phase, the join results are scattered over different nodes; therefore, during the Merge phase, the scattered join results are collected and merged.



```
SELECT * FROM FT, DT1, DT2
WHERE FT.FK1 = DT1.PK1 AND FT.FK2 = DT2.PK2 AND DT2.PK2 = 'b1'
```

Dimension Table 1(DT1)

PK1	D1
a_0	$D_{1,1}$
a_1	$D_{1,2}$
a_2	$D_{1,3}$

Fact Table(FT)

FK1	FK2	F
a_0	b_1	F0
a_1	b_1	F1
a_2	b_0	F2
a_0	b_0	F3

Dimension Table 2(DT2)

PK2	D2
b_0	$D_{2,1}$
b_1	$D_{2,2}$

DT2
FT
DT1
Filter Phase
<(b1, 1, "=", "D"), (D2,2)>
BF
Mapper
<(b1, 1, "=", "D"), (D2,2)>
Mapper
<(b1, 1, "=", "F"), (a0, b1, F0)>
<(b1, 1, "=", "F"), (a1, b1, F1)>
<(a1, 0, "=", "F"), (a1, b1, F1)>
<(a0, 0, "=", "F"), (a0, b1, F0)>
Mapper
<(a0, 0, "=", "D"), (D1,1)>
<(a1, 0, "=", "D"), (D1,2)>
<(a2, 0, "=", "D"), (D1,3)>
Reducer
<(a0, b1), (D2,2, F0)>
<(a1, b1), (D2,2, F1)>
Reducer
<(a0, b1), (D1,1, F0)>
<(a1, b1), (D1,2, F1)>
Merge Phase
<(a0, b1), (D1,1, D2,2, F0)>
<(a1, b1), (D1,2, D2,2, F1)>

Figure 2. An example of TJSGM

III. Performance Evaluation

Experiments are conducted for evaluating the performance of our proposed method. The experimental environment consists of four nodes connected with a gigabit Ethernet switch. Each node is equipped with a 2.66GHz Quad CPU, 4GB RAM and 500GB HDD, and runs Apache Hadoop 0.20.2 [9] on Ubuntu 10.04 LTS operating system. In total, the master node can invoke six Mappers and six Reducers on the other three slave nodes. Each Mapper/Reducer has 1GB memory for running its Map/Reduce task.

The experiments adopt the Star Schema Benchmark [7]. The benchmark uses four dimension tables and one fact table, and introduces four types of query benchmarks, namely Q1.x, Q2.x, Q3.x and Q4.x. The numbers in the types indicate the number of dimension tables for joining. For example, Q2.x is a query type with restrictions on two dimensions [7]. The experimental data is generated by the generator of the benchmark with the data size up to 11.3 GBytes.

Two experiments are conducted. We implement another method SGM proposed in [6] to compare the performance with TJSGM and another method TJSGM- for evaluating the effect of our partition function. In the experiments, we extend the join operators in the benchmark queries to anyone of the operators =, <, >, ≤, ≥ and ≠.

In the first experiment, the performance of SGM and TJSGM is illustrated in Figure 3. The experiment compares the performance under the equi-join operations, since SGM supports the equi-join operations only. In Figure 3, the X-axis is the introduced query types of the Star Schema Benchmark, and the Y-axis is the execution time in seconds. According to the experiment, even if TJSGM uses more data for supporting theta-join operations, the performance of TJSGM is similar to that of SGM. TJSGM performs even better in some query types of high data selectivity (i.e., more data satisfying the filter conditions) of dimension tables such as types Q2.1, Q3.1, and Q4.1. This is because the manipulation of keys on data pairs in TJSGM makes highly related data pairs (i.e. data pairs for joining) located together on the same node. However, SGM does not make highly related data pairs close but scattered over nodes, which leads to higher join overhead. In fact, the manipulation on keys of our method is the very idea that makes theta-join workable on the MapReduce platform.

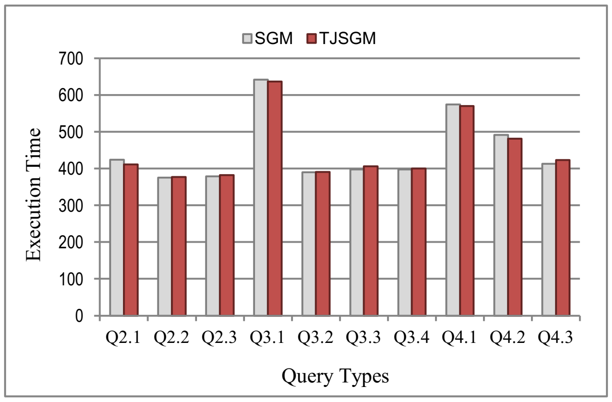


Figure 3. Comparison of SGM and TJSGM

For evaluating the effect of our partition function, we implement another method TJSGM-, which uses the same algorithms as TJSGM except that TJSGM- adopts the default partition function of MapReduce. Figure 4 illustrates the result. The Y-axis in Figure 4 is the execution time in seconds. The performance of TJSGM is 59% faster than that of TJSGM-.

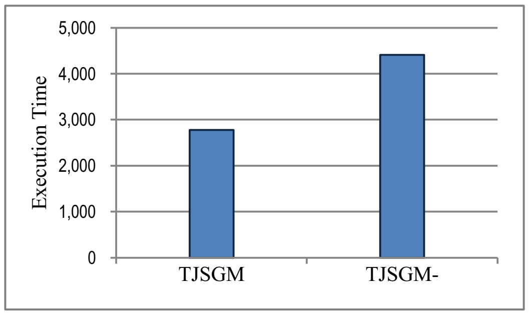


Figure 4. Comparison of TJSGM and TJSGM-

IV. CONCLUSIONS AND FUTURE WORK

We propose a method for executing star joins with the theta-join operations on the MapReduce framework. The method offers a novel manipulation on keys and a partition function to match up the MapReduce paradigm. According to the experimental results, our method achieves similar performance to SGM and is even better in some cases, but our method supports more join-query types. Our future work includes extending the idea of key manipulation and partition functions to other data processing applications, such as data mining.

REFE RENCES

[1] A. Abelló, J. Ferrarons and O. Romero, "Building Cubes with MapReduce," in Proceedings of Data Warehouse and Online Analytical Processing, pp.17–24 , 2011.

[2] F. N. Afrati and J. D. Ullman, "Optimizing Joins in a Map-Reduce Environment," in Proceedings of International Conference on Extending Database Technology, pp. 99–110 , 2010.

[3] J. Chandar, "Join Algorithms using Map/Reduce," http://www.inf.ed.ac.uk/publications/thesis/online/IM100859.pdf, 2010.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–150, 2004.

[5] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google File System," in Proceedings of the 9th ACM Symposium on Operating Systems Principles, pp. 29–43, 2003.

[6] H. Han, H. Jung, H. Eom and H. Y. Yeom, "Scatter-Gather-Merge: An Efficient Star-Join Query Processing Algorithm for Data-Parallel Frameworks," Cluster Computing, Vol. 14, No. 2, pp. 183–197, 2010.

[7] P. O'Neil, B. O'Neil and X. Chen, "Star Schema Benchmark," http://www.cs.umb.edu/~poneil/StarSchemaB.PDF , 2007.

[8] H. C. Yang and R. L. Hsiao, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters." in Proceedings of the 6th Symposium on Operating System Design and Implementation, pp.1029–1040, 2004.

[9] Apache Hadoop, http://hadoop.apache.org/, 2007.